AFRL-RY-WP-TR-2008-1262

A FRAMEWORK FOR RETARGETING RADIO DESIGNS

Dr. Gary J. Minden, Dr. Joseph B. Evans, Dr. W. Perry Alexander, Ed Komp, and Garrin Kimmel

The University of Kansas Center for Research, Inc.

AUGUST 2008
Final Report

AIR FORCE RESEARCH LABORATORY
SENSORS DIRECTORATE
WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7320
AIR FORCE MATERIEL COMMAND
UNITED STATES AIR FORCE

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report was cleared for public release by the Defense Advanced Research Projects Agency (DARPA) and is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RY-WP-TR-2008-1262 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH THE ASSIGNED DISTRIBUTION STATEMENT.

*//signature//

ALFRED SCARPELLI
Project Engineer
Advanced Sensor Components Branch
Aerospace Components and Subsystems
Technology Division

//signature//

BRADLEY J. PAUL
Chief, Advanced Sensor Components Branch
Aerospace Components and Subsystems
Technology Division
Sensors Directorate

//signature//

TODD A. KASTLE
Chief, Aerospace Components and Subsystems
Technology Division
Sensors Directorate

This report is published in the interest of scientific and technical information exchange and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

*Disseminated copies will show "//signature//" stamped or typed above the signature blocks.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
August 2008	Final	10 August 2007 – 30 August 2008

4. TITLE AND SUBTITLE
A FRAMEWORK FOR RETARGETING RADIO DESIGNS

5a. CONTRACT NUMBER
FA8650-07-C-7733

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER
69199F

6. AUTHOR(S)
Dr. Gary J. Minden, Dr. Joseph B. Evans, Dr. W. Perry Alexander, Ed Komp, and Garrin Kimmel

5d. PROJECT NUMBER
ARPS

5e. TASK NUMBER
ND

5f. WORK UNIT NUMBER
ARPSNDBL

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
The University of Kansas Center for Research, Inc.
Information and Telecommunications Technology Center
2385 Irving Hill Road
Lawrence, KS 66045

8. PERFORMING ORGANIZATION REPORT NUMBER
ITTC-FY2009-TR-48370-01

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Air Force Research Laboratory
Sensors Directorate
Wright-Patterson Air Force Base, OH 45433-7320
Air Force Materiel Command
United States Air Force

DARPA/STO
3701 Fairfax Drive
Arlington, VA 22203-1714

10. SPONSORING/MONITORING AGENCY ACRONYM(S)
AFRL/RYDI

11. SPONSORING/MONITORING AGENCY REPORT NUMBER(S)
AFRL-RY-WP-TR-2008-1262

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

13. SUPPLEMENTARY NOTES
PAO case number: DISTAR 12837; date cleared: 23 Feb 2009. This report contains color.

14. ABSTRACT
We address the process of designing software defined radios. Our goal is to design radio functions once and use automated tools to transform the design to implementations on different platforms. ReTarget is an approach and a process that (1) describes radio functions in a specification language, Rosetta, (2) translates specifications to an intermediate language suitable for hardware and software implementation, and (3) translates the intermediate language to VHDL and C programs. In this feasibility study, we demonstrated specifying radio functions in a high level language, translation to an intermediate language, and further translation to an implementation language. A number of issues arose during the study. Specifically, (1) what mechanisms, that are compatible with both hardware and software implementation, should be used to exchange information between radio functions and (2) how to set up control and management of radio function in this primarily data-flow oriented domain.

15. SUBJECT TERMS
software defined radio, system level design, flexible radio design, agile radios, dynamic spectrum access, automated design, Rosetta

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT:	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON (Monitor)
a. REPORT	b. ABSTRACT	c. THIS PAGE			Alfred J. Scarpelli
Unclassified	Unclassified	Unclassified	SAR	66	19b. TELEPHONE NUMBER *(Include Area Code)* N/A

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39-18

Table of Contents

Section **Page**

List of Figures

Figure **Page**

1 Executive Summary

This work addresses the design of software-defined radios. In particular, we address the challenge of re-using radio designs to implement multiple physical radios using multiple implementation fabrics.

Software defined radios (SDRs) offer the capability of quickly changing the capabilities of a radio by re-programming the hardware platform. The Joint Tactical Radio System (JTRS) [1] is the acme of sophisticated SDRs. However, the approach taken in the JTRS program is to define a Software Communications Architecture (SCA) [2] that primarily specifies the interfaces between system components rather than the components themselves. And while this approach promises re-usable software components, it does not address the challenge of re-using designs mapped to future radio hardware components. Our focus is on retargeting a common radio design to multiple implementation platforms.

We completed a feasibility study for a Framework for ReTargeting Radio Designs to facilitate the design, re-use, and re-targeting of radio designs to multiple hardware platforms. By "radio design" we mean the specification and implementation of the system(s) that actually process bits and signals that implement the physical radio. By "framework" we mean a set of tools to specify, optimize, and instantiate a radio design in a number of technologies (e.g., general purpose processors, digital signal processors, field programmable gate arrays, or application specific integrated circuits). And by "retargeting" we mean the re-use of radio designs on multiple hardware implementations.

Such a design framework is vital within the context of ongoing DARPA and DoD programs because these programs focus on quick development of low-cost radios as technology advances, as mission demands increase, and as system cost becomes a major concern. A major focus of current DARPA radio programs is to reduce the cost of flexible radios. This feasibility study supports that focus by addressing the challenge of reducing the non-recurring engineering costs of designing radios.

Our approach to the Radio Design Framework, called ReTarget, is threefold. First, the framework uses a standards based specification language to describe radio components. Second, we implemented an intermediate description language, called dlang, that is suitable for automated processing. The intermediate language supports well defined transforms and validation checks based on user criteria, technology capabilities, and constraints. Third, we implemented transforms from dlang into the commercially available languages VHSIC Hardware Description Language (VHDL) and C. (VHSIC stands for Very High Speed Integrated Circuits, a Department of Defense program in the early 1980's.) Commercial tools can be used to carry out the final design implementation.

Our initial study focuses on designing the architecture for ReTarget and demonstrating the feasibility of the architecture. A significant effort was invested in the intermediate description language. The language needed to be able to capture radio functions, needed to be sufficiently flexible to represent software and hardware system, and be capable of supporting trade-offs between different alternative implementations. We implemented a few example radio functions, implemented one means to transfer information between radio functions, and generated VHDL and C implementations.

Our experience with this feasibility study of ReTarget affirms our convection that tools for automated design of radios from specifications is possible, but that significant attention in the future needs to be paid to how radio functions exchange information, how one controls radio functions, and how one transforms a specification to meet implementation goals.

This work was supported by The Defense Advanced Research Projects Agency (DARPA) and The Air Force Research Laboratory (AFRL) under contract FA8650-07-C-7733.

2 Introduction

ReTarget is an application-oriented specification language, compiler, and simulator. The language expresses dataflow applications at high abstraction levels. This class of application includes those for radio communication processing and digital signal processing. The functional nature of the language is important for two reasons. First, users easily express radio designs at an abstract level with dataflow paradigms. Second, functional languages are amenable to symbolic and automatic manipulation to (a) convert from high abstraction levels to implementation levels, (b) optimize conversions based on different criteria and target architectures, and (c) elicit through formal methods properties concerning an application and their implementation. A key characteristic of ReTarget is the inclusion of formal specifications of program properties, the capability to use those formal specifications during the compilation process to meet user established criteria, and the ability to make specific statements about the properties of programs and implementations.

Radio designs are specified in a high-level, formal specification language. The language is amenable to automated transforms that can take into account user criteria (e.g., low power, high speed, or small size) and implementation constraints (e.g., implementation in a specific field programmable gate array). The transform mechanism will be based on provided and user defined rule sets. That is, users will be able to control and manage the process by defining transformation rules.

In order to take these concepts to practice, future example radio designs will need to be specified with the user interface, transformed using the transformation mechanism and design rules and implemented on selected radio platforms subject to DARPA program needs. In addition to technology research and development, this preliminary effort will create a roadmap for implementation of the framework, including identification of development challenges and metrics.

3 Feasibility Study Implementation

In our feasibility study, we focused on three activities for our framework:

1. specify typical radio functions,
2. express radio functions and intra-communications in an intermediate language, and
3. translate the intermediate language into executable code.

Figure 1 illustrates the ReTarget concept. Specifications are written in Rosetta. Once written, we expect specifications to be re-used multiple times. Specifications are translated to an intermediate language. For this feasibility study, we developed a functional intermediate language called dlang. Intermediate forms expressed in dlang can also be re-used. The dlang intermediate forms are then translated to implementation languages. In this study we generated C and behavioral VHDL. We developed dlang as a functional language because we anticipate that future work will entail applying transforms to dlang forms to achieve specific implementation goals and functional languages facilitate applying transforms.

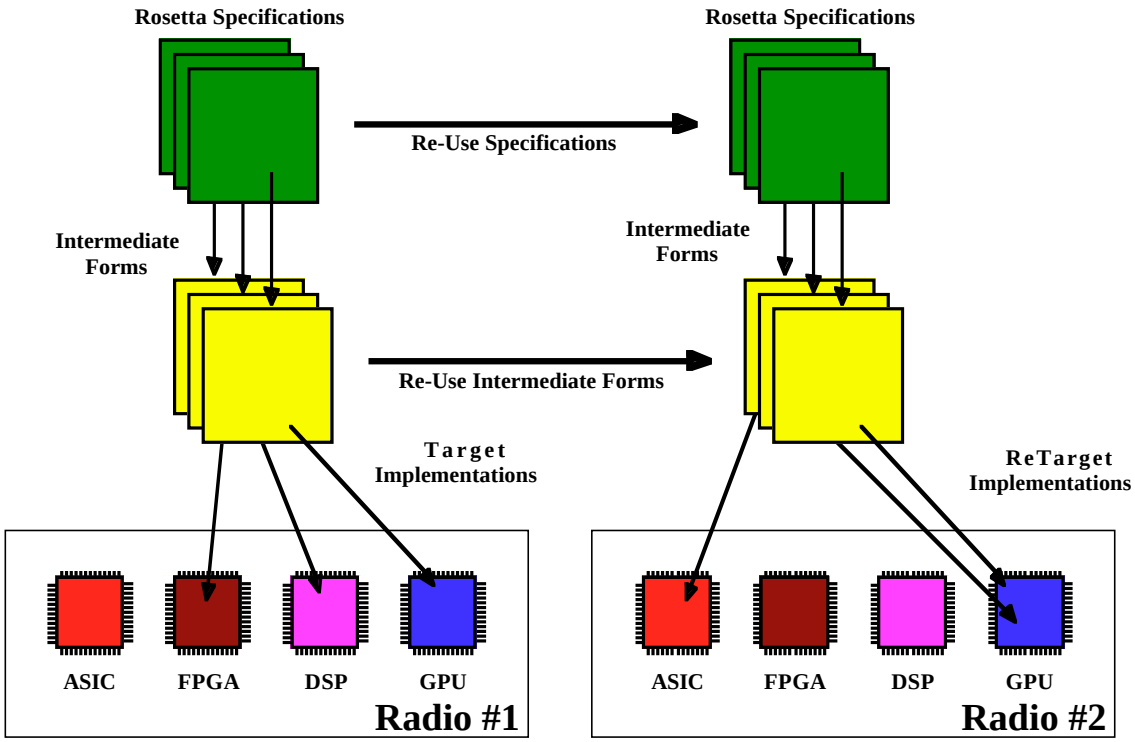


Figure 1 The ReTarget Concept

3.1 Rosetta Specification Language

We selected the Rosetta [3] specification language for ReTarget. Rosetta is an emerging IEEE standard (IEEE Standard P1699) and the principal author is a co-investigator on this project.

Rosetta is a systems-level design language. Thus it is structured in a way that different experts can describe the behavior of their aspect of a system and then compose those multiple aspects into a system-level design. For example, one expert would design the logic or algorithm of a circuit. A second expert would describe the power of the logic function. A third expert would describe the physical size of the logic function. Rosetta enables these independent descriptions and the combination of these descriptions into a system level design.

Rosetta is a specification language. Thus, when we write (taken from a part of the Advanced Encryption Standard (AES) specification):

```
// Define utility function subBytesRow usign explicit sequence creation
    subBytesRow(a::rowType)::rowType is
      [sbox(bv2nat(a(0))),
       sbox(bv2nat(a(1))),
       sbox(bv2nat(a(2))),
       sbox(bv2nat(a(3)))];
```

we are not describing how you compute `subBytesRow` but are describing what `subBytesRow` is equivalent to. Rosetta describes the relationships and equivalences between statements about the design. Because Rosetta explicitly states relationships and equivalences, automated means, tools, and techniques can be used to check the consistency of a collection of Rosetta statements.

Rosetta specifications are translated into an abstract syntax tree (AST) using existing tools [9][10]. The AST is translated into a functional language designed to express radio functions and the exchange of information between radio functions. We call this intermediate functional language dlang.

3.2 The Intermediate Language dlang

We developed an intermediate language, called dlang, for describing radio functions. dlang is a functional language with extensions for computational effects (such as state and concurrency). In the functional subset of the language, all expressions are pure: that is, they simply describe how to transform inputs (arguments) to outputs. The state and concurrency extensions are structured using monads, which encapsulate computational effects. Using monads simplifies the reasoning and transformation of those computations in the compilation toolset.

An example we commonly use is the following:

1. You have the expression “(* X 5)” meaning you multiply X by 5.

2. You reduce the expression to "(ConstMultiply X 5)" meaning you recognize the constant "5" and use a more efficient constant coefficient multiplier.
3. You reduce the expression to "(Add X (ShiftLeft X 2))" meaning you recognize that multiplication by "5" can be accomplished by a shift of X and an addition.

A significant complication in the specification and expression languages (i.e. Rosetta and dlang) is how radio functions communicate. We address this issue below. dlang is described in detail in Section 4.1.

3.3 Implementation Languages

We focused on translating simple dlang expressions to the languages behavioral VHDL and C. These languages were chosen because they represent paths to field programmable gate arrays (FPGAs), application specific integrated circuits (ASICs), digital signal processors (DSPs), and general purpose processors (GPPs). Within ReTarget you can use the same specification and dlang description of a radio function and generate either VHDL or C language expressions depending on your targeted implementation.

3.4 Examples

In this section we present samples of radio function descriptions in Rosetta and dlang. Out intent is to provide a "flavor" of these languages and not a complete description or example. We present a complete example in Appendix B. These languages are designed to support well-defined expressions and automated analysis and transformations. Expressions in Rosetta and dlang are "different" from most programming languages. In Rosetta, we state equivalences, not assignment. In dlang, we state how radio functions interconnect. Each language is selected for precision in description and automated processing. In our framework, engineers would describe radio functions in Rosetta and not need to look at dlang or resulting programs.

3.4.1 Examples of Rosetta

A key radio function is modulation of a carrier signal. In a mathematical sense we use the expression:

$$S(t) = A(t) \times Cos(\omega(t)t + \phi(t)).$$

In this case A(t) represents the amplitude of the signal as it varies over time, ω(t) represents the change in frequency over time and ϕ(t) represents the change in phase over time. Our primary modulation techniques are derived from this expression. For Amplitude Modulation (AM), we hold ω(t) and ϕ(t) constant and change A(t). For Frequency Modulation (FM) we hold A(t) and ϕ(t) constant and change ω(t). For phase modulation we hold A(t) and ω(t) constant and change ϕ(t).

In Rosetta we describe the general modulation function as:

```
// Constant values

    twoPi :: real is 2*pi;

    modulate( am, fm, ps :: real; f, t :: real) :: real is
      am*cos(twoPi*(f+fm)*t+ps);
```

This Rosetta expression states that “twoPi” is a real value and equivalent to 2*π. It also states the relationship between the term “modulate” and its description: amplitude (“am”) modulated cosine with frequency “f,” frequency offset “fm,” time “t,” and phase offset “ps.”

Now, when we want to define an AM modulator, we use the expression:

```
amMod[T::type](k::<*(x::T)::real*>; f::real; t::real; s::T)::real is
      modulate( k(s),0.0,0.0,f,t);
```

In essence, this states that AM modulator “amMod” is a “modulate” item with function “k(s)” as the amplitude, no frequency offset, and no phase offset. Function “k(s)” takes a symbol of type T and maps it to a real. When using “amMod” the type T is inferred from its usage environment. T is thus a placeholder in the definition of “amMod” and “amMod” can be used to generate continuous wave (CW), amplitude shift keying (ASK), and amplitude modulated (AM) signals by providing different instantiations of “k(s).”

Further examples are provided in Appendices B and C.

3.4.2 Examples of dlang

dlang is a functional language. In general, that means there is no memory (variables) and no side-effects in the language. This facilitates automated tools manipulating or transforming language statements.

The following is a partial listing of a handcoded Huffman [8] encoder in dlang.

```
(data HuffmanTree (Emit char)
                  (Node HuffmanTree HuffmanTree))
;

;;  Decoder traverses the HuffmanTree based on the input bitstream,
;;    until it encounters an Emit node, corresponding to a
;;    received symbol.
;;  Accepts input as a bit stream
;;  Outputs decoded symbol (bits 8)

(define (decode (fulltree (HuffmanTree a)) (tree (HuffmanTree a))
                (monad [(@inChan (react (Msg b)))
                        (@outChan (react (Msg a)))] Unit))
   (case tree
     ((Emit a)   (do (signal @outChan (ReqSend (Just a)))
                   (decode fulltree fulltree [@inChan @outChan])))
     ((Node l r) (do (val <- (recvMsg [@inChan]))
                   (case val
                         ((Just v) (case v
                                     ((True) (decode fulltree r
                                                     [@inChan @outChan]))
                                     ((False) (decode fulltree l
                                                      [@inChan @outChan]))))
                         ((Nothing) (do (signal @outChan
                                                (ReqSend Nothing))
                                        (return Unit)))))))))
```

The “data” statement defines a “HuffmanTree” as either an “Emit” node with value type “char” or a “Node” with left and right branches of types “HuffmanTree.” This is a conventional binary tree. The idea is that as bits arrive for decoding the program walks down the tree, going left for 0 and right for 1, until encountering an “Emit” node. When the “Emit” node is encountered, the value attached to the “Emit” node is returned.

The description of the Huffman decoder in a functional language, like dlang, is a bit more complex. Because functional languages generally do not support state or variables, items like tables and trees must be passed from one invocation to the next. In the case of the Huffman decoder, a full “HuffmanTree” and a partial “HuffmanTree” must be passed to each invocation of the “decode” function. Also passed in the invocation of “decode” are input and output channels described in the “monad” statement. Input and output channels will be discussed below.

When invoked, “decode” looks at the partial tree passed as the argument “tree.” If “tree” is a node of type “Emit,” the value of the “Emit” node, “a,” is output with the “signal” statement and the decoder is invoked again to receive the next encoded symbol. If “tree” is of type “Node” with a left and right branch, then we attempt to read a value from the input channel. There are two possibilities, a value is available, denoted “(Just v)” or nothing is available, i.e. the upstream process has not yet produced a value, denoted “(Nothing).” When a value is received, it is used to determine which branch of the “Node” to use for further decoding. If the value is “True” the right, or “r” tree is used. If the value is “False” the left, or “l” tree is used. Decoding continues until an “Emit” node is encountered and decoding of the next encoded symbol begins. When “(Nothing)” is

received on the input, “Nothing” is sent on the output. Inter-function communication will be discussed below.

While dlang is somewhat readable, it is an intermediate form for automated processing. We do not expect engineers to write dlang programs and to rarely expect them to look at dlang programs. dlang outputs from the Rosetta processor are like a compiler intermediate form and are consumed directly by downstream processing tools.

3.4.3 Example of Generated Code

The following C function was generated from the Huffman decode description. As automatically generated code, it is not necessarily “pretty.”

```
Unit decode (HuffmanTree fulltree,HuffmanTree tree,Msg inChan,Msg outChan)
  {HuffmanTree var0 = tree ;
   switch ((int)((var0) [0]))
     {case 0 :
        {PTR a = (PTR)((var0) [1]) ;
         {Maybe var1 = (Maybe)(malloc (8)) ;
          ((var1) [0]) = ((PTR)(1));
          ((var1) [1]) = ((PTR)(a));
          Msg var2 = (Msg)(malloc (8)) ;
          ((var2) [0]) = ((PTR)(1));
          ((var2) [1]) = ((PTR)(var1));
          dlangSignal ((Msg)(outChan),(Msg)(var2));
          return (decode
                   ((HuffmanTree)(fulltree),
                    (HuffmanTree)(fulltree),
                    (Msg)(inChan),
                    (Msg)(outChan)));}}
      case 1 :
        {HuffmanTree l = (HuffmanTree)((var0) [1]) ;
         HuffmanTree r = (HuffmanTree)((var0) [2]) ;
         {Maybe val = recvMsg ((Msg)(inChan)) ;
          Maybe var3 = val ;
          switch ((int)((var3) [0]))
            {case 1 :
               {PTR v = (PTR)((var3) [1]) ;
                {Bool var4 = v ;
                 switch ((int)((var4) [0]))
                   {case 1 :
                      {return (decode
                                ((HuffmanTree)(fulltree),
                                 (HuffmanTree)(r),
                                 (Msg)(inChan),
                                 (Msg)(outChan)));}
                    case 0 :
                      {return (decode
                                ((HuffmanTree)(fulltree),
                                 (HuffmanTree)(l),
                                 (Msg)(inChan),
                                 (Msg)(outChan)));}}}}
             case 0 :
               {{Msg var5 = (Msg)(malloc (8)) ;
                 ((var5) [0]) = ((PTR)(1));
                 ((var5) [1]) = ((PTR)(&tag_Nothing));
                 dlangSignal ((Msg)(outChan),(Msg)(var5));
                 return (&tag_Unit);}}}}}}}
```

4 ReTarget System Architecture

ReTarget is implemented as a set of programs written in Haskell [6]. Haskell is a non-strict functional language with a strong type system used to implement Rosetta tools. The existing Rosetta toolset, consisting of Raskell and InterpreterLib, translates Rosetta to an Abstract Syntax Tree (AST). We implemented a new program to translate the AST to dlang. The ReTarget program flow is shown in Figure 2.

We have implemented a Rosetta plug-in for the Eclipse [7] development environment. The Eclipse Rosetta plug-in supports keyword highlighting, a syntax checker, and a standard interface to Rosetta tools. Additional program flow steps, now executed manually, can be included in the plug-in in the future.

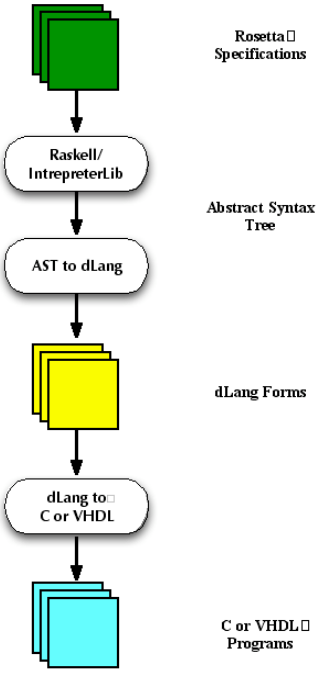


Figure 2 The Program Flow from a Rosetta Specifications to Final C or VHDL Program

Rosetta [3], C [4], and VHDL [5] are defined elsewhere. In the next section we will describe dlang and provide a rationale for designing and implementing a new intermediate language.

4.1 The dlang Intermediate Language

There are three major elements of dlang:

1. Pure functional language
2. Monadic Imperative state
3. Monadic Reactive Concurrency

The “pure functional language” is important because we can do unfold and fold transforms that allow us to take advantage of the inherent parallelism (modulo data dependencies) of functional languages. The example Hamming encoder is an example of this - we define a specification that is a set of list comprehensions. The specification is independent of the actual size of the data input, yet when we go to synthesize the circuit, we can unfold the definitions of the comprehensions to get multiple instantiations of a component (a parity checker, in the case of the Hamming encoder), which can be executed in parallel.

Being able to perform program manipulations is critical if we are to target both hardware and software from a single descriptive source. The difference between hardware and software design is in mechanisms for the designer to precisely control the number of computational resources that are available. In software, you have no choice but to use the resources offered by the CPU, so consequently you end up time-multiplexing data parallelism across those limited resources, generally using some sort of loop. On the other hand, in hardware you can dictate exactly how many instances of a given computational resource you wish to use, and incorporate that number into your circuit.

The (sequential) software model and the (parallel) hardware model are really two different views on the same computation, it is just that the view of the computation in software is from the time perspective, while the hardware view is from the space, or structural, perspective. Because dlang is a pure functional language, these two views are compatible, as there are no implicit computational effects in a pure functional program. This means that a wide variety of transformations are possible, and safe, because computational effects can be ignored. It is possible to take a pure dlang program and inline it into its primitive components, resulting in a large combinational circuit. This is unwise from a design perspective because there is no accounting for space or time limitations in the target model. Consequently, dlang includes two constructs for controlling the time and space behavior exhibited by a pure program.

First, a dlang program consists of a set of top-level function definitions, along with a distinguished main expression that calls the defined functions. A top-level function delimits a shared resource. The body of the function will be implemented as a single circuit, regardless of the number of calls to the function. If there are, in fact, multiple calls to the function, the function block contains arbitration logic that will process those calls in some (undetermined) sequential order. This is a space/time tradeoff. Because the pure subset of dlang does not have side effects, a designer can take a single shared function block, duplicate it, and distribute the calls to the original function amongst the various duplicated blocks.

Second, dlang includes a “let” construct that introduces sequence and sharing into a program. Suppose that a particular dlang expression will result in a very long critical path. Inserting a pipeline register into the circuit, and thus reducing that critical path, is as simple as selecting a sub-expression and adding a named binding via a let expression. Likewise, a common sub-expression can be factored into a single let binding. Rather than a circuit for each instance of the sub-expression, the synthesis scheme will generate a single circuit that can be shared among all references to that sub-expression.

The ability to perform these transformations gives a justification to the pure functional subset of dlang. However, the pure functional subset of the language is simply too restrictive to be practical for building radios, or embedded applications in general. This is because the pure functional model is heavily skewed towards dataflow computation, which is a poor abstraction when constructing control-intensive or reactive systems, both of which are common traits of embedded systems. Control implies a notion of state. This, along with the interaction with external entities implied by reactive components, forces dlang to include a mechanism for performing both stateful and reactive effects. However, we wish to add constructs for building these sorts of computations without breaking the purity that we rely on to justify transformations that allow us to target both hardware and software. Therefore, we use monads to structure effective computations.

By structuring computational effects using monads, we get a static delineation between pure operations and effective computations that is expressed at the type-level. A monadic computation can include pure computations, but the converse is not true. This means that the extent that transformations are valid are clearly delimited by the monadic encapsulation of effects. The synthesis toolset is free to transform programs within a monadic computation, but the transformations may never escape those monad boundaries.

What this means for radio components is that there is a fairly regular structure. A component is a loop that receives data from some extra entity via the reactive concurrency construct, performs some processing on that data, sends the data onto another component via the reactive concurrency construct, and repeats. The component may perform several loop iterations between receives/sends. The Hamming encoder is an example of this. Also, the component may carry some state across iterations. The primary purpose of this state is to simplify the specification of control logic.

The dlang monadic constructs are segmented into two categories: two for performing imperative effects, and one for reactive concurrency. The imperative, or state, constructs include a “get” function for accessing state, and a “put” function for mutating the state. Both of these functions use addressable state. The semantics of the state monadic constructs are as one would expect.

The reactive concurrency construct is called “signal.” The signal construct takes a message (or request) and routes that message to an external entity. From the point of view of a component, the outside world is accessible only via the signal construct, and can only be affected by signaling a request. Once a computation signals a request, the computation will block until a response is returned from an external entity. Again, signaling a request and then interpreting the response is the only way to observe the outside world from a reactive computation. That is the totality of the dlang concurrency semantics within the language. Obviously, the semantics leaves much undefined -- such as what is the content of the request/response messages, how are they interpreted, and how are they routed between components.

We elected not to extend dlang with special semantics for different types of reactive communication, simply because the possible range of different inter-component communication is boundless. As examples, the inspiration for this work used the reactive monad construct to model an operating systems kernel, while in contrast we have defined

a series of point-to-point communication behaviors that include a mailbox, bounded FIFO, and unbounded FIFO. The basic concurrency constructs allow us to model each of these, but the specific behavior for each communication is defined within dlang as a collection of functions, rather than as special language constructs. These functions are arranged according to a regular pattern; we have developed a nomenclature to refer to the various elements.

First, a thread is a processing component. A thread may consist of a combination of pure functional, imperative, and reactive expression. The single point of interaction with outside entities for a thread is via the reactive constructs. However, a thread may interact with several different external entities, with each interaction accomplished with a different protocol. A named interaction point for a thread is called a port. Finally, the ports are connected to a service, which is a collection of dlang functions that implements the protocol the connected threads use.

For example, consider a component that performs stream processing: consuming data from an input source, manipulating the data, than sending the data on to an output sink. Figure 3 has two examples of this architecture. The boxes labeled “source,” “transform,” and “sink” are dlang threads. Each thread has ports, indicated by the grayed semicircles where arrows enter and exit the thread. The rectangles labeled “F” in the top figure and “Bus” in the bottom are services. In the top figure, the “transform” figure has two ports, each of which is connected to a separate service. In the bottom figure, the “transform” component has a single port, connected to the “bus” service.

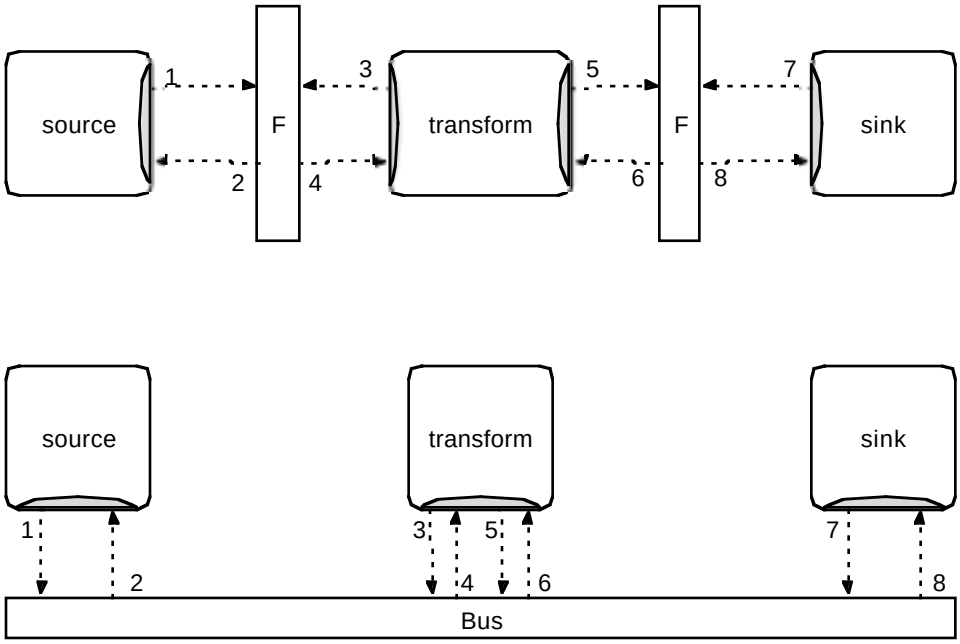


Figure 3 Two Mechanisms for Inter-Function Communications

The transmission of messages (requests/responses) between threads and services is indicated in Figure 3 by the dotted arrows. The arrows are numbered to indicate the

ordering of messages. For example, in the top figure, the “source” would (1) issue a Send request to the “F” service that would include the data it wishes to send to the “transform” thread. The thread would block until the service (2) responded with an Ack response. The “transform” component (3) issues a Receive request to the service “F,” and then block until “F” (4) sends a response that includes the data originally transmitted by “source”. The series of messages (5,6,7,8) uses the same protocol with “transform” acting the sender and “sink” the receiver.

It is critical to note that the ordering of messages is for expository purposes only. Because the threads “source,” “transform,” and “sink” are operating concurrently, each may issue a request at any time, subject the protocol restriction. Temporally, there will be an ordering between request/response pairs: (1 comes before 2), (3 before 4), and so on. In the dlang concurrency architecture, a request/response pair is called a service transaction. The logic of a thread will also dictate the ordering of service transactions -- for example, the transaction (3,4) necessarily comes before (5,6), since it is necessary for the “transform” service to receive data to process before processing it and sending it on to the “sink” thread.

The lower figure has all three threads connected via a single “Bus” service. Consequently, the “transform” thread has a single port for communicating with the Bus service, in contrast to the two ports in the point-to-point model with FIFO services. Moreover, the reorganization of the system architecture requires each thread to add additional information to the requests that issued to the Bus service indicating the intended destination for the message. In contrast, the architecture in the top diagram allows communication addressing to be implied by the port.

The “F” and “Bus” services in the diagrams implement a given protocol. This is accomplished with a pair of dlang functions: a handler function that takes a request issued by a thread and generates the appropriate response, and a scheduler function that determines when a response generated by the handler can be returned to the appropriate blocked thread. These functions include imperative effects that allow the service to implement the control portion of its protocol. This can be illustrated using the top example architecture from above. The service that handles communication between the “source” and “transform” thread will initially be in a state waiting for a request from either thread. Upon receiving a request, the service invokes the handler function on the request. If the request is from “source” and is a Send request, the service will store the value included in the request and generate an Ack response. The scheduler function will then be invoked, which will return that Ack response to the source thread. Alternatively, if the request is a Receive request from the “transform” thread, the handler will note that a receive is pending, but cannot generate a response immediately, because there is no previous send request from the source thread. In this case, the scheduler function will indicate that there are no pending responses to transmit to threads, and the scheduler will wait for the next request, which can only come from the “source” thread, as the “transform” thread will remain blocked until the service returns a response to its receive request.

Hence, the service implements a very simple mailbox protocol: a receive request will cause the issuing thread to block until there has been a matching send requests. Moreover, if a thread issues two consecutive send requests, and the service does not

receive an intervening receive request, the sending thread will block on the second send until a Receive is issued. This is a degenerate case of a bounded FIFO, which would allow an arbitrary (but fixed for a particular instance) number of sends without an intervening receive, simply by using a larger imperative state for buffering sent data. The bounded FIFO is, in turn, a specialization of an unbounded FIFO. However, the unbounded FIFO requires that the service be able to dynamically allocate storage, which is typically not possible in a direct hardware implementation. This suggests a dlang design process: a system is modeled as a collection of thread communicating with each other via a given protocol. The system developer then generates a service definition that provides the loosest bounds on implementation, such as the unbounded FIFO implementation of point-to- point communication, and uses that model for simulation and early design testing. As threads are mapped to various implementation targets, technology specific knowledge is used to drive the transformations of dlang thread definitions to get implementations that can match the capabilities of the target platform.

4.2 Comparison of dlang to other Hardware Description Languages

In this section we compare dlang to other languages engineers have used to describe hardware and software systems.

4.2.1 VHDL and Verilog

VHDL [5] and Verilog [11] are the primary of hardware description languages used by engineers. Both of them exploit parallelism inherent in hardware. However, there is an enormous burden on the engineer to deal with low-level details of communication between concurrent components. Inter-component communication is managed by incorporating protocol details into the component that is being designed. Contrast this with dlang, where all external communication is transaction-based. The thread has to be aware of the protocol up to the point of knowing the correct sequence of requests and responses, but the implementation of the protocol - at the level of clocked signals - is irrelevant from the point of view of the thread. Similarly, because the handler and scheduler functions for a service are written in dlang, these implementation themselves can ignore low-level details. All of the precise details of thread to service communication and service to thread communication are handled by the synthesis routines. Moreover, this synthesis model is such that it can be both for handling shared function blocks for the pure subset of the language as well as thread to/from service communication. Moreover, the type system of dlang is more sophisticated than VHDL, and much more than that of Verilog. Using algebraic data types, dlang can model data at a high level, and the compilation and synthesis routines can compile the high-level data representations to efficient implementations.

4.2.2 Lava, Hawk, and Ruby

Lava [12], Hawk [13], and Ruby [14] follow in a tradition of modeling circuits as functional programs. We will focus on Lava, since we are most familiar with it, however, the analysis is relevant to all three. Lava is a language that is embedded in Haskell, which uses Haskell's lazy lists to model synchronous circuits as streams of bits. Circuits are built structurally, and can be simulated (directly within Haskell), synthesized (to structural VHDL), or verified (using an external Satisfiable-based solver). Compared to

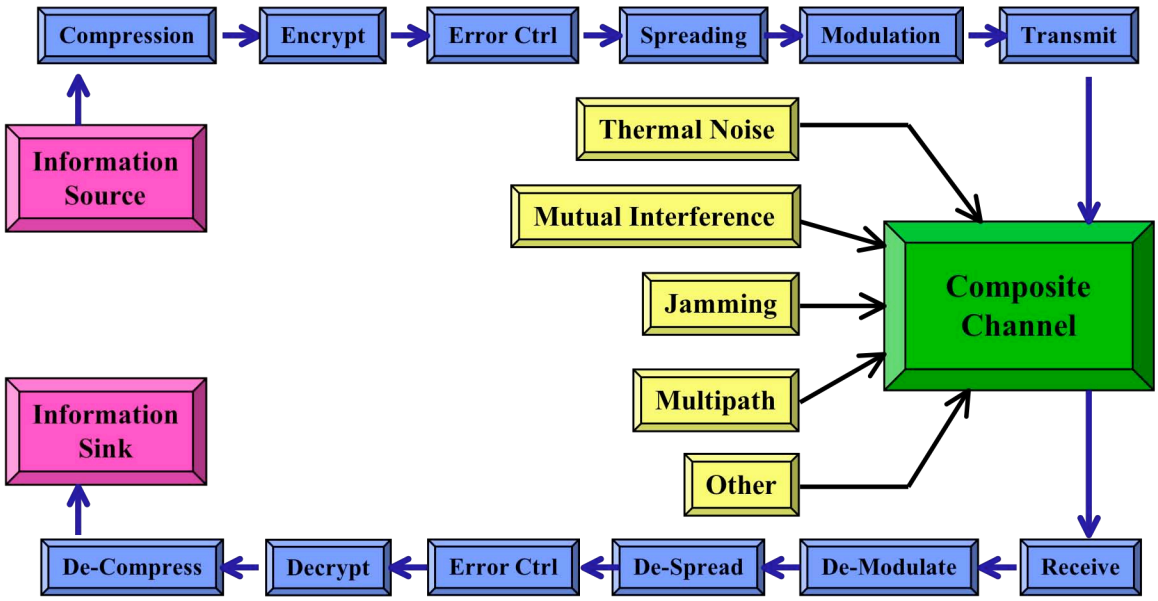
dlang, the circuit representations are much lower-level, both in data representation (all data has to be represented as bits) and computationally (state can only be introduced using a delay element, which models a flip-flop). Moreover, there is no language-level model of concurrency, such as dlang's signal construct.

4.2.3 SAFL

dlang is most closely related to, and derivative of the Statically Allocated Functional Language (SAFL) [15]. Many of the ideas, both in the language design and in the synthesis implementation, reflect these origins. The most important departure is the formal basis of monads to model computation in dlang. To a great degree, dlang simply provides a convenient toolset for constructing and manipulating formal monadic models of systems. We chose to develop our own toolset, rather than reuse SAFL, for pragmatic reasons: it is unclear as to the availability of the SAFL toolset and the intellectual property restrictions on its use.

5 Inter-Function Communications

Typically, when we describe software-defined radios (SDR), we focus on the processing functions and not so much on how information is passed between functions. Figure 4 is a typical SDR block diagram. We are interested in the behavior and performance of these processing functions. However, the manner in which processing functions are connected is crucial to the performance and capabilities of a SDR.



Every processing stage is programmable and controllable.

Figure 4 A Typical Software-Defined Radio Processing Flow

The JTRS program selected the Common Object Request Broker Architecture (CORBA) as the means to interconnect radio functions. While flexible, CORBA uses significant processor capacity that leads to increased component costs and energy use. CORBA is a software middleware system. Adapting CORBA to hardware, such as FPGAs, ASICs, or DSPs has been difficult. Another approach is to use shared memory to store intermediate values and pass pointers to data structures in shared memory among the components of a SDR. This works well for software radio functions implemented in DSPs or GPPs and can work for radio functions implemented in FPGAs or ASICs.

A key result of our work is the recognition that the means to move information around a SDR is almost as important as the radio functions needed to implement a SDR. Inter-function communications must be as rigorously specified as radio functions. Figure 5 attempts to capture this concept. Moving data between functions is fundamental to implementing SDRs and engineers invest a significant design effort to ensure data transfer is flawless.

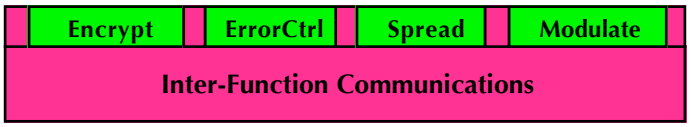


Figure 5 The Inter-Function Communications between Radio Functions

In our framework, we explicitly specify the inter-function communication mechanism. For this feasibility study, we use a simple mechanism. Data passed between functions are represented by a pair: < Status, Value >. Value is a bit vector holding the information. Status indicates if the Value is valid or not. The states of Status are: Invalid, Valid, and EndOfBlock. This abstraction works for software and hardware defined radio functions.

For example, in hardware we commonly use a central clock to coordinate activity in multiple radio functions. We can think that on each clock pulse, a radio function outputs < Status, Value > pair. The receiving function takes action based on Status. If Status is Valid, the Value is processed. If Status is Invalid, the Value is not processed. If Status is EndOfBlock, special processing might take place.

The same abstraction can be used in software implementations of radio functions. Each radio function is an execution thread in the software implementation. Threads are executed when data is Valid. For this study, we implemented a message-passing algorithm.

We arrived at this abstraction after considering a range of radio function blocks and experience in designing, implementing, and reviewing SDRs. We know, from experience, that a compression function will have more symbols, e.g., < Status, Value > pairs coming in than are output. We know that an error control function will generate more output bits than input bits. We know that a spreading function will output more chips than input bits. Because radio functions have different input/output rates, we use the abstraction described above to define how radio functions communicate and use a common clock to keep functions synchronized.

Our framework allows multiple definitions of the inter-function communication process. That is, the definition of how data is transferred between radio functions is defined in the Rosetta and dlang system. Other inter-function communication processes can be designed. Our intent is to separate the design of processing functions from the design of inter-function communications.

6 Results

In this feasibility study we implemented a framework for retargeting radio designs. We accomplished the following:

1. Wrote radio function specifications in a standard, systems level specifications language, Rosetta.
2. Translated Rosetta specifications to an intermediate language, dlang, using existing and new tools.
3. Translated dlang intermediate forms to C and VHDL using new tools.
4. Implemented an inter-function communications capability based on message passing defined in dlang and suitable for implementation in both hardware and software.

6.1 Lessons Learned

1. Writing a coherent set of specifications is more difficult than anticipated. Most radio functions are described as algorithms. Moving descriptions from algorithms and mathematical equations to specifications was harder than anticipated. There are multiple ways of writing radio function specifications and determining the best way, if there is one, requires further work.
2. Inter-function communications is critical and needs to be explicitly specified. SDR design systems must include mechanisms to implement different inter-function communications systems.

6.2 Conclusions

We demonstrated a prototype capability for writing radio functions in a specification language and translating specifications to an implementation language through an intermediate functional language. We recognized the needs to (a) consider specifying how information is exchanged between radio-functions and (b) refine the radio design domain for systems level design frameworks.

7 Future

This project has only started to explore the issues of automated radio design. During our research we identified the following as important topics to research in the future.

1. Rosetta is a powerful specification language aimed at addressing a wide range of system level design problems. The full range of Rosetta capabilities needs to be refined through defining a Radio Domain to make expressing radio functions easier for the radio designer. That is, we need to develop a Radio Specification Language within the Rosetta system. Rosetta explicitly supports writing such domains, thus a framework is already in place.
2. Further work on abstracting, defining, and implementing inter-function communications is needed. In particular we need to find abstractions that work for both hardware and software implementations and enable the goal of quickly retargeting designs to different implementations. In our feasibility study, we defined inter-function communication in dlang. We need to move those definitions to Rosetta.
3. This project did not work with the control and management of radio functions. We only worked with the processing functions. Many functions, like the Huffman compressor/decompressor, AES encryption/decryption, and direct sequence spreaders need to be initialized before execution. Future work needs to address how control and management functions are defined and integrated with the radio functions.
4. dlang is designed to make it easy to transform one statement into another. We did not implement any transforms or work on a transform system. A transform system and appropriate transform rules for multiple implementation targets needs to be designed, implemented, and tested.

8 References

[1] D. Bauman, "Joint Tactical Radio System, Introductory Remarks for Media Teleconference," Joint Tactical Radio System Joint Program Executive Office, May 3, 2006.

[2] "Software Communications Architecture Specification, MSRC-5000SCA, V2.2," Joint Tactical Radio System Joint Program Office, November 17, 2001.

[3] W. P. Alexander, System-Level Design with Rosetta, Morgan Kaufmann Publishers, Inc, 2006, ISBN: 9781558607712.

[4] B. W. Kernighan, and D. M. Ritchie, C Programming Language, 2nd ed., Prentice Hall PTR, 1988.

[5] P. J. Ashenden, The Designer's Guide to VHDL. 3rd ed., Morgan Kaufmann, 2008.

[6] P. Hudak, The Haskell School of Expression: Learning Functional Programming through Multimedia, Cambridge University Press, 2000.

[7] S. Holzner, Eclipse, O'Reilly Media, Inc., 2004.

[8] D. A. Huffman, "A Method for the Construction of Minimum-Redundancy Codes", Proceedings of the I.R.E., pp 1098-1102, September 1952.

[9] P. Weaver, G. Kimmell, and W. P. Alexander, “Software Engineering with Algebra Combinators,” in Proceedings of the ACM International Conference on Generative Programming and Component Engineering (GPCE'07), October 1-3, 2007.

[10] E. Komp, G. Kimmell, J. Ward, and W. P. Alexander, “The Raskell Evaluation Environment, Technical Report,” The University of Kansas Information and Telecommunications Technology Center, 2335 Irving Hill Rd, Lawrence, KS, USA, November 2003.

[11] “Standard Verilog Hardware Description Language Reference Manual, ”Institute of Electrical and Electronic Engineers, New York, NY, 2007.

[12] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh, “Lava: Hardware design in Haskell,” in Proceeding of The International Conference on Functional Programming, Baltimore, Maryland, September 26-29, 1998, pp. 174-184.

[13] J. Matthews, B. Cook, and J. Launchbury, “Microprocessor specification in Hawk,” in Proceedings of the 1998 International Conference on Computer Languages, Chicago, IL, May 14-16, 1998, pp. 90-101.

[14] G. Jones and M. Sheeran. “Designing Arithmetic Circuits by Refinement in Ruby,” Proceedings of the Second international Conference on Mathematics of Program Construction (June 29 - July 03, 1992). R. S. Bird, C. Morgan, and J. Woodcock, Eds. Lecture Notes In Computer Science, vol. 669. Springer-Verlag, London, 208-232..

[15] R. Sharp, Higher-Level Hardware Synthesis, SpringerVerlag, 2004

Appendix A – External Presentations

We made the presentations concerning Radio Design to the following groups during the project:

DARPA Kick-off meeting, October 2-3, 2007, Arlington, VA.

DARPA WNaN Project Review at MA/COM, December 5, 2007, Lowell, MA.

IDGA 6th Annual Software Radio Summit, February 26, 2008, Vienna, VA.

Rockwell-Collins, February 13, 2008, Cedar Rapids, IA.

DoD/Finnish Workshop, March 10-11, 2008, Washington D.C.

Rockwell-Collins, April 30, 2008, Cedar Rapids, IA.

Microsoft Research, June 2008.

JTRS Science & Technology Forum (JSTeF), September 17-18, 2008, San Diego, CA.

Formal Design Languages Conference, October 2008, Stugart, Germany.

Professor Evans is participating in a STTR project sponsored by the JTRS office.

Professor Minden is a member of a National Research Council (NRC) panel on "Universal Radio Frequency Systems for Special Operations Forces."

Appendix B – Example of Huffman Encoder

This appendix presents a complete example of using Rosetta and dlang to generate executable code. We use the Huffman [8] compression/de-compression function as an example. The reasons for using the Huffman compressor are that it is a possible radio function and it has different input and output data rates.

B.1 Rosetta Specification for Huffman Compressor

```
package huffman() :: state_based is

    // Cleanup: Record Fields should be lowercase
    // Cleanup: Types should be uppercase
  HuffmanTree :: type is data
     Emit(HuffmanTreeVal :: char)::isEmit |
     Node(Left :: HuffmanTree; Right :: HuffmanTree)::isNode
  end data;

  Pair(a :: type; b :: type) :: type is data
     Pair (first :: a; second :: b) :: isPair
  end data;

  // Table(a :: type; b :: type) :: type is sequence (Pair(a,b));

  treeToTable (bs :: sequence(bit); tree :: HuffmanTree) ::
    Table(char, sequence(bit)) is
    if (isEmit (tree)) then
      [ Pair( HuffmanTreeVal(tree), bs) ]
    else
      treeToTable((bs + [0]), Left(tree))  +
      treeToTable((bs + [1]), Right(tree))
    end if;

  lookup [ a :: type; b :: type] (key :: a; table :: Table(a,b)) ::
b is
     if ( first(head(table)) == key) then
       second(head(table))
     else
       lookup(key, tail(table))
     end if;

  facet huffmanDecoder (  inChan :: channel bit ;
                       fulltree :: design HuffmanTree ;
                       outChan ::  channel char) :: state_based is

     StateName :: type is [waitInput, decoding, checkOutput, done];

     currentState :: StateName;
     tree :: HuffmanTree;

   begin

//  init_label:
//    tree' = fulltree and
//    currentState' = waitInput;

  input_label: (currentState = waitInput) =>
     inChan'Receive and currentState' = decoding;

  // We have input and we're now deciding what to do with it.
  decoding: (currentState = decoding) =>
     (tree' = if (inChan == 1) then Right(tree)
                               else Left(tree) end if)
    and currentState' = checkOutput ;
```

```
  // If we're ready to output, then do so.  Otherwise just wait.
  output_label: (currentState = checkOutput) =>
       currentState' = waitInput;

  output_emit: (currentState = checkOutput) and isEmit(tree) =>
       (outChan'Send(HuffmanTreeVal(tree)))
       and tree' = fulltree
       and currentState' = waitInput;

  end facet huffmanDecoder;

  facet huffmanEncoder  ( inChan :: channel char;
                      fulltree :: design HuffmanTree;
                      outChan :: channel bit) :: state_based is

     StateName :: type is [waitInput , lookup, sending, done];

     currentState :: StateName;
     outdata :: sequence(bit);
     table :: Table is treeToTable(fulltree);

  begin

  init_label:  currentState' = waitInput;

  input_label: (currentState = waitInput) =>
     inChan'Receive and currentState' = lookup;

  lookup_label:  (currentState = lookup) =>
       outdata' = lookup (SigVal(input), table)
       and currentState' = sending ;

  send_label: (currentState = sending) and (outdata = []) =>
       currentState' = waitInput;

  send_notempty: (currentState = sending) and (not (outdata = [])) =>
                  (outChan'Send(head(outdata)))
                      and  outdata' = tail(outdata)
                and currentState' = sending;

end facet huffmanEncoder;

end package huffman;
```

B.2 dlang description of the Huffman Compressor

This description was hand generated for testing the dlang to C and VHDL translators.

```
;;(import Pt2PtKernel)
(import Pt2PtFifoKernel)

(data HuffmanTree [a] (Emit a)
                      (Node HuffmanTree HuffmanTree))
;

;;;  Encoder uses a table lookups.
;;;  input symbol (bits 8) -> variable length list of bits for encoding
;;;  Accepts input symbol by symbol
;;;  Outputs encoded value bit by bit
(define (encode (tree (HuffmanTree a))
         (monad [(@inChan (react (Msg a)))
                 (@outChan (react (Msg b)))] Unit))
        (do (val <- (recvMsg [@inChan]))
            (case val
                ((Just x) (do (putList (lookup x (treeToTable tree))
                                       [@outChan])
                              (encode tree [@inChan @outChan])))
                ((Nothing) (do (signal @outChan (ReqSend Nothing))
                               (return Unit))))
            ))

;;;  Decoder traverses the HuffmanTree based on the input bitstream,
;;;    until it encounters an Emit node, corresponding
;;;    to a received symbol.
;;;  Accepts input as a bit stream
;;;  Outputs decoded symbol (bits 8)
(define (decode (fulltree (HuffmanTree a)) (tree  (HuffmanTree a))
                (monad [(@inChan (react (Msg b)))
                        (@outChan (react (Msg a)))] Unit))
       (case tree
             ((Emit a)   (do (signal @outChan (ReqSend (Just a)))
                             (decode fulltree fulltree
                                     [@inChan @outChan])))
             ((Node l r) (do (val <- (recvMsg [@inChan]))
                             (case val
                              ((Just v) (case v
                                          ((True) (decode fulltree r
                                                          [@inChan @outChan]))
                                                ((False) (decode fulltree l
                                                                 [@inChan @outChan])))))
                                   ((Nothing) (do (signal @outChan
                                                          (ReqSend Nothing))
                                                  (return Unit))))))))

; ============================================================
;    Utilities / Support
; ============================================================

 (define (recvMsg (monad [(@chan (react (Msg a)))] (Maybe a)))
   (do (resp <- (signal @chan ReqRecv))
       (case resp ((RspRecv val) (return val)))))


; Helper function to push the elements of a list to an
;   output stream in order.
```

```
(define (putList (lst (List a))
                 (monad [(@chan (react (Msg a)))] Unit))
         (case lst
             ((Cons x xs) (do (signal @chan (ReqSend (Just x)))
                              (putList xs [@chan])
                              (return Unit)))
             ((Null)      (return Unit))))

(define (lookup (key keyTy) (table (List (* keyTy valTy))) valTy)
    (case table
       ((Cons hd tail)
            (case (= (prj 0 hd) key)
                ((True)  (prj 1 hd))
                ((False)  (lookup key tail))))
    ))

(define (treeToTable (tree (HuffmanTree a)) (List (* a (List Bool))))
   (treeToTableHelper Null tree))

(define (treeToTableHelper (bs (List Bool)) (tree (HuffmanTree a))
                           (List (* a (List Bool))))
   (case tree
       ((Emit bv) (Cons (tuple bv bs) Null))
       ((Node l r) (concat (treeToTableHelper (concat bs (Cons False
Null)) l)
                           (treeToTableHelper (concat bs (Cons True
Null)) r)))))

; ============================================================
;    Tests
; ============================================================

;;; Definition of a very simple Huffman Tree for test purposes.

(define (tree1 (HuffmanTree (bits 8)))
        (Node (Node (Node (Emit 0b00000001)
                          (Emit 0b00000011))
                    (Emit 0b00000010))
              (Node (Emit 0b00000100)
                    (Emit 0b00000111))))

;;; Data generator for test purposes.
(define (src (monad [(@chan (react (Msg (bits 8))))] Unit))
    (do (signal @chan (ReqSend (Just 0b00000001)))  ;;  1 : 3 bits out
        (signal @chan (ReqSend (Just 0b00000111)))  ;;  7 : 2 bits  5
        (signal @chan (ReqSend (Just 0b00000010)))  ;;  2 : 2 bits  7
        (signal @chan (ReqSend (Just 0b00000100)))  ;;  4 : 2 bits  9
        (signal @chan (ReqSend (Just 0b00000001)))  ;;  1 : 3 bits 12
        (signal @chan (ReqSend (Just 0b00000111)))  ;;  7 : 2 bits 14
        (signal @chan (ReqSend (Just 0b00000010)))  ;;  2 : 2 bits 16
        (signal @chan (ReqSend (Just 0b00000111)))  ;;  7 : 2 bits 18
        (signal @chan (ReqSend (Just 0b00000011)))  ;;  3 : 3 bits 21
        (signal @chan (ReqSend (Just 0b00000011)))  ;;  3 : 3 bits 24
        (signal @chan (ReqSend Nothing))
        (return Unit)))

;;; Accumulates received data in a list for test purposes.
```

```
(define (sink (acc (List a)) (monad [(@chan (react (Msg a)))] (List a)))
    (do (v <- (recvMsg [@chan]))
        (case v
            ((Nothing) (return acc))
            ((Just x)  (sink (Cons x acc) [@chan])))))

;
;  This configuration defines a very simple system:
;
;  Src -> Encoder -> Decoder -> Sink
;
;  Between each two computational blocks is a Point-to-Point
;   (Stop and Wait) service.
;  There must be exactly one sender and one receiver attached
;   to each service.
;  If Send (Receive) request arrives, it waits until a matching
;   Receive (Send) Request
;  is present, at which time both a response is generated
;   for both requests.

(configuration huffmanPt2Pt
  (service @src2encoder (Msg (bits 8)) msgHandler dequeue initKernelC
continueHandler)
  (service @decoder2sink  (Msg (bits 8)) msgHandler dequeue initKernelC
continueHandler)
  (service @encoder2decoder  (Msg Bool) msgHandler dequeue initKernelC
continueHandler)
  (thread (src [@src2encoder]))
  (thread (sink Null [@decoder2sink]))
  (thread (encode (tree1) [@src2encoder @encoder2decoder]))
  (thread (decode (tree1) (tree1) [@encoder2decoder @decoder2sink]))
  )

(configuration test
  (thread (lookup  0b00000001 (Cons (tuple 0b00000001 True) Null)))
)
```

B.3 Automatically generated dlang description of the Huffman Compressor from Rosetta Specification

This description was automatically generated for testing the Rosetta to dlang and dlang to C and VHDL translators. Being automatically generated, the formatting is pretty.

```
(data HuffmanTree (Emit char)
                  (Node HuffmanTree HuffmanTree))
(data Pair [a b] (Pair a b))

(define (treeToTable (bs (List bit)) (tree HuffmanTree) (Table char
(List bit)))
  (case (isEmit tree [])
   (True (Cons (Pair (HuffmanTreeVal tree []) bs []) Nil))
   (False (+ (treeToTable (+ bs (Cons 0 Nil)) (Left tree []) [])
(treeToTable (+ bs (Cons 1 Nil)) (Right tree []) [])))))

(define (lookup (key a) (table (Table a b)) b)
  (case (== (first (head table []) []) key [])
   (True (second (head table []) []))
   (False (lookup key (tail table []) []))))

(define (huffmanDecoder (fulltree HuffmanTree)
   (monad [(@inChan (react (Msg bit)))
           (@outChan (react (Msg char)))
          (@inChan_state (state Null bit))
          (@outChan_state (state Null char))
          (@currentState_state (state Null StateName))
          (@tree_state (state Null HuffmanTree))] Null))

  (do (do (currentState <- (get @currentState_state Unit))
       (do (tree <- (get @tree_state Unit))
        (do (inChan <- (get @inChan_state Unit))
         (do (outChan <- (get @outChan_state Unit))
          (return Unit)))))
  (do (do (case (= currentState waitInput [])
             (True (do (do (inChan_response <- (signal @inChan Receive))
                           (put @inChan_state Unit inChan_response))
                       (do (put @currentState Unit decoding) (return
Unit))))
             (False (return Unit)))
           (do (case (= currentState decoding [])
                 (True (do (put @tree Unit (case (== inChan 1 [])
                                             (True (Right tree []))
                                                   (False (Left tree []))))
                           (do (put @currentState Unit checkOutput)
(return Unit))))
                 (False (return Unit))) (do (case (= currentState
checkOutput [])
                                                  (True (do (put
@currentState Unit waitInput) (return Unit)))
                                                  (False (return Unit)))
                                            (do (case (and (= currentState
checkOutput []) (isEmit tree []) [])
                                                  (True (do (signal
@outChan (Send (HuffmanTreeVal tree []))))
                                                        (do (put @tree Unit
fulltree) (do (put @currentState Unit waitInput) (return Unit)))))
                                                  (False (return Unit)))
                                                (return Unit)))))
      (huffmanDecoder fulltree [@inChan @outChan @inChan_state
@outChan_state])))))
```

```
(define (huffmanEncoder (fulltree HuffmanTree)
           (monad [(@inChan (react (Msg char)))
           (@outChan (react (Msg bit)))
                   (@inChan_state (state Null char))
                   (@outChan_state (state Null bit))
                   (@currentState_state (state Null StateName))
                   (@outdata_state (state Null (List bit)))] Null))

  (do (do (currentState <- (get @currentState_state Unit))
       (do (outdata <- (get @outdata_state Unit))
        (do (inChan <- (get @inChan_state Unit))
         (do (outChan <- (get @outChan_state Unit))
             (return Unit)))))
      (do (do (case (= currentState waitInput [])
                (True (do (do (inChan_response <- (signal @inChan
Receive))
                              (put @inChan_state Unit inChan_response))
                          (do (put @currentState Unit lookup) (return
Unit))))
                (False (return Unit)))
              (do (case (= currentState lookup [])
                        (True (do (put @outdata Unit (lookup (SigVal
input []) table []))
                                  (do (put @currentState Unit sending)
(return Unit))))
                        (False (return Unit)))
                  (do (case (and (= currentState sending []) (=
outdata Nil []) [])
                            (True (do (put @currentState Unit waitInput)
(return Unit)))
                            (False (return Unit)))
                      (do (case (and (= currentState sending []) (not
(= outdata Nil []) []) [])
                                (True (do (signal @outChan (Send (head
outdata [])))
                                          (do (put @outdata Unit (tail
outdata []))
                                              (do (put @currentState Unit
sending) (return Unit)))))
                                (False (return Unit))) (return Unit)))))
           (huffmanEncoder fulltree [@inChan @outChan @inChan_state
@outChan_state])))))
```

B.4 Automatically generated C description of the Huffman Compressor from hand-coded dlang

This description was automatically generated for testing the Rosetta to dlang and dlang to C and VHDL translators. Being automatically generated, the formatting is pretty.

```
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <stdarg.h>
#include "dlangCore.h"
typedef int BitVector;
typedef void** PTR;
typedef void** AThread;
typedef void** Bool;
typedef void** HuffmanTree;
typedef void** KernelState;
typedef void** List;
typedef void** Maybe;
typedef void** Msg;
typedef void** Unit;
void* tag_AThread = (void*) 0;
void* tag_Cons = (void*) 0;
void* tag_Emit = (void*) 0;
void* tag_False = (void*) 0;
void* tag_Just = (void*) 1;
void* tag_KS = (void*) 0;
void* tag_Node = (void*) 1;
void* tag_Nothing = (void*) 0;
void* tag_Null = (void*) 1;
void* tag_ReqRecv = (void*) 0;
void* tag_ReqSend = (void*) 1;
void* tag_RspAck = (void*) 2;
void* tag_RspRecv = (void*) 3;
void* tag_True = (void*) 1;
void* tag_Unit = (void*) 0;
List append (List l1,List l2);
List concat (List l1,List l2);
Unit encode (HuffmanTree tree,Msg inChan,Msg outChan);
Unit decode (HuffmanTree fulltree,HuffmanTree tree,Msg inChan,Msg
outChan);
Maybe recvMsg (Msg chan);
Unit putList (List lst,Msg chan);
PTR lookup (PTR key,List table);
List treeToTable (HuffmanTree tree);
List treeToTableHelper (List bs,HuffmanTree tree);
HuffmanTree tree1 ();
Unit src (Msg chan);
List sink (List acc,Msg chan);
Unit initKernelC (int numthreads,void** kstate);
Maybe dequeue (int dummy,void** kstate);
Unit msgHandler (int tid,int chan,Msg msg,void** kstate);
Unit handleSend (int sendTid,Msg msg,void** kstate);
Unit handleRecv (int recvTid,Msg msg,void** kstate);
Unit scheduleSendRecv (int sendTid,Msg sendMsg,int recvTid,void** kstate);
List consEOL (PTR value,List lst);
int main ();
List append (List l1,List l2)
  {List var0 = l1 ;
   switch ((int)((var0) [0]))
     {case 1 :
        {return (l2);}
```

```
      case 0 :
        {PTR hd = (PTR)((var0) [1]) ;
         List tail = (List)((var0) [2]) ;
         {List var1 = (List)(malloc (12)) ;
          ((var1) [0]) = ((PTR)(0));
          ((var1) [1]) = ((PTR)(hd));
          ((var1) [2]) = ((PTR)(append ((List)(tail),(List)(l2))));
          return (var1);}}}}
List concat (List l1,List l2)
  {return (append ((List)(l1),(List)(l2)));}
Unit encode (HuffmanTree tree,Msg inChan,Msg outChan)
  {Maybe val = recvMsg ((Msg)(inChan)) ;
   Maybe var0 = val ;
   switch ((int)((var0) [0]))
     {case 1 :
        {PTR x = (PTR)((var0) [1]) ;
         {putList ((List)(lookup ((PTR)(x),(List)(treeToTable
((HuffmanTree)(tree))))),(Msg)(outChan));
          return (encode
((HuffmanTree)(tree),(Msg)(inChan),(Msg)(outChan)));}}
      case 0 :
        {{Msg var1 = (Msg)(malloc (8)) ;
          ((var1) [0]) = ((PTR)(1));
          ((var1) [1]) = ((PTR)(&tag_Nothing));
          dlangSignal ((Msg)(outChan),(Msg)(var1));
          return (&tag_Unit);}}}}
Unit decode (HuffmanTree fulltree,HuffmanTree tree,Msg inChan,Msg outChan)
  {HuffmanTree var0 = tree ;
   switch ((int)((var0) [0]))
     {case 0 :
        {PTR a = (PTR)((var0) [1]) ;
         {Maybe var1 = (Maybe)(malloc (8)) ;
          ((var1) [0]) = ((PTR)(1));
          ((var1) [1]) = ((PTR)(a));
          Msg var2 = (Msg)(malloc (8)) ;
          ((var2) [0]) = ((PTR)(1));
          ((var2) [1]) = ((PTR)(var1));
          dlangSignal ((Msg)(outChan),(Msg)(var2));
          return (decode
((HuffmanTree)(fulltree),(HuffmanTree)(fulltree),(Msg)(inChan),(Msg)(outCh
an)));}}
      case 1 :
        {HuffmanTree l = (HuffmanTree)((var0) [1]) ;
         HuffmanTree r = (HuffmanTree)((var0) [2]) ;
         {Maybe val = recvMsg ((Msg)(inChan)) ;
          Maybe var3 = val ;
          switch ((int)((var3) [0]))
            {case 1 :
               {PTR v = (PTR)((var3) [1]) ;
                {Bool var4 = v ;
                 switch ((int)((var4) [0]))
                   {case 1 :
                      {return (decode
((HuffmanTree)(fulltree),(HuffmanTree)(r),(Msg)(inChan),(Msg)(outChan)));}
                    case 0 :
```

```
                      {return (decode
((HuffmanTree)(fulltree),(HuffmanTree)(l),(Msg)(inChan),(Msg)(outChan)));}
}}}
             case 0 :
               {{Msg var5 = (Msg)(malloc (8)) ;
                 ((var5) [0]) = ((PTR)(1));
                 ((var5) [1]) = ((PTR)(&tag_Nothing));
                 dlangSignal ((Msg)(outChan),(Msg)(var5));
                 return (&tag_Unit);}}}}}}}
Maybe recvMsg (Msg chan)
  {Msg resp = dlangSignal ((Msg)(chan),(Msg)(&tag_ReqRecv)) ;
   Msg var0 = resp ;
   switch ((int)((var0) [0]))
     {case 3 :
        {Maybe val = (Maybe)((var0) [1]) ;
         return (val);}}}
Unit putList (List lst,Msg chan)
  {List var0 = lst ;
   switch ((int)((var0) [0]))
     {case 0 :
        {PTR x = (PTR)((var0) [1]) ;
         List xs = (List)((var0) [2]) ;
         {Maybe var1 = (Maybe)(malloc (8)) ;
          ((var1) [0]) = ((PTR)(1));
          ((var1) [1]) = ((PTR)(x));
          Msg var2 = (Msg)(malloc (8)) ;
          ((var2) [0]) = ((PTR)(1));
          ((var2) [1]) = ((PTR)(var1));
          dlangSignal ((Msg)(chan),(Msg)(var2));
          {putList ((List)(xs),(Msg)(chan));
           return (&tag_Unit);}}}
      case 1 :
        {return (&tag_Unit);}}}
PTR lookup (PTR key,List table)
  {List var0 = table ;
   switch ((int)((var0) [0]))
     {case 0 :
        {PTR hd = (PTR)((var0) [1]) ;
         List tail = (List)((var0) [2]) ;
         {Bool var1 = ((PTR)((hd) [1])) == (key) ? &tag_True : &tag_False
;
          switch ((int)((var1) [0]))
            {case 1 :
               {return ((PTR)((hd) [2]));}
             case 0 :
               {return (lookup ((PTR)(key),(List)(tail)));}}}}}}
List treeToTable (HuffmanTree tree)
  {return (treeToTableHelper ((List)(&tag_Null),(HuffmanTree)(tree)));}
List treeToTableHelper (List bs,HuffmanTree tree)
  {HuffmanTree var0 = tree ;
   switch ((int)((var0) [0]))
     {case 0 :
        {PTR bv = (PTR)((var0) [1]) ;
         {void** var1 = (void**)(malloc (12)) ;
          ((var1) [0]) = ((PTR)(0));
          ((var1) [1]) = ((PTR)(bv));
          ((var1) [2]) = ((PTR)(bs));
```

```
          List var2 = (List)(malloc (12)) ;
          ((var2) [0]) = ((PTR)(0));
          ((var2) [1]) = ((PTR)(var1));
          ((var2) [2]) = ((PTR)(&tag_Null));
          return (var2);}}
       case 1 :
         {HuffmanTree l = (HuffmanTree)((var0) [1]) ;
          HuffmanTree r = (HuffmanTree)((var0) [2]) ;
          {List var3 = (List)(malloc (12)) ;
           ((var3) [0]) = ((PTR)(0));
           ((var3) [1]) = ((PTR)(&tag_False));
           ((var3) [2]) = ((PTR)(&tag_Null));
           List var4 = (List)(malloc (12)) ;
           ((var4) [0]) = ((PTR)(0));
           ((var4) [1]) = ((PTR)(&tag_True));
           ((var4) [2]) = ((PTR)(&tag_Null));
           return (concat ((List)(treeToTableHelper ((List)(concat
((List)(bs),(List)(var3))),(HuffmanTree)(l))),(List)(treeToTableHelper
((List)(concat ((List)(bs),(List)(var4))),(HuffmanTree)(r)))));}}}}
HuffmanTree tree1 ()
  {HuffmanTree var0 = (HuffmanTree)(malloc (8)) ;
   ((var0) [0]) = ((PTR)(0));
   ((var0) [1]) = ((PTR)(1));
   HuffmanTree var1 = (HuffmanTree)(malloc (8)) ;
   ((var1) [0]) = ((PTR)(0));
   ((var1) [1]) = ((PTR)(3));
   HuffmanTree var2 = (HuffmanTree)(malloc (12)) ;
   ((var2) [0]) = ((PTR)(1));
   ((var2) [1]) = ((PTR)(var0));
   ((var2) [2]) = ((PTR)(var1));
   HuffmanTree var3 = (HuffmanTree)(malloc (8)) ;
   ((var3) [0]) = ((PTR)(0));
   ((var3) [1]) = ((PTR)(2));
   HuffmanTree var4 = (HuffmanTree)(malloc (12)) ;
   ((var4) [0]) = ((PTR)(1));
   ((var4) [1]) = ((PTR)(var2));
   ((var4) [2]) = ((PTR)(var3));
   HuffmanTree var5 = (HuffmanTree)(malloc (8)) ;
   ((var5) [0]) = ((PTR)(0));
   ((var5) [1]) = ((PTR)(4));
   HuffmanTree var6 = (HuffmanTree)(malloc (8)) ;
   ((var6) [0]) = ((PTR)(0));
   ((var6) [1]) = ((PTR)(7));
   HuffmanTree var7 = (HuffmanTree)(malloc (12)) ;
   ((var7) [0]) = ((PTR)(1));
   ((var7) [1]) = ((PTR)(var5));
   ((var7) [2]) = ((PTR)(var6));
   HuffmanTree var8 = (HuffmanTree)(malloc (12)) ;
   ((var8) [0]) = ((PTR)(1));
   ((var8) [1]) = ((PTR)(var4));
   ((var8) [2]) = ((PTR)(var7));
   return (var8);}
Unit src (Msg chan)
  {Maybe var0 = (Maybe)(malloc (8)) ;
   ((var0) [0]) = ((PTR)(1));
   ((var0) [1]) = ((PTR)(1));
   Msg var1 = (Msg)(malloc (8)) ;
```

```
((var1) [0]) = ((PTR)(1));
((var1) [1]) = ((PTR)(var0));
dlangSignal ((Msg)(chan),(Msg)(var1));
{Maybe var2 = (Maybe)(malloc (8)) ;
 ((var2) [0]) = ((PTR)(1));
 ((var2) [1]) = ((PTR)(7));
 Msg var3 = (Msg)(malloc (8)) ;
 ((var3) [0]) = ((PTR)(1));
 ((var3) [1]) = ((PTR)(var2));
 dlangSignal ((Msg)(chan),(Msg)(var3));
 {Maybe var4 = (Maybe)(malloc (8)) ;
  ((var4) [0]) = ((PTR)(1));
  ((var4) [1]) = ((PTR)(2));
  Msg var5 = (Msg)(malloc (8)) ;
  ((var5) [0]) = ((PTR)(1));
  ((var5) [1]) = ((PTR)(var4));
  dlangSignal ((Msg)(chan),(Msg)(var5));
  {Maybe var6 = (Maybe)(malloc (8)) ;
   ((var6) [0]) = ((PTR)(1));
   ((var6) [1]) = ((PTR)(4));
   Msg var7 = (Msg)(malloc (8)) ;
   ((var7) [0]) = ((PTR)(1));
   ((var7) [1]) = ((PTR)(var6));
   dlangSignal ((Msg)(chan),(Msg)(var7));
   {Maybe var8 = (Maybe)(malloc (8)) ;
    ((var8) [0]) = ((PTR)(1));
    ((var8) [1]) = ((PTR)(1));
    Msg var9 = (Msg)(malloc (8)) ;
    ((var9) [0]) = ((PTR)(1));
    ((var9) [1]) = ((PTR)(var8));
    dlangSignal ((Msg)(chan),(Msg)(var9));
    {Maybe var10 = (Maybe)(malloc (8)) ;
     ((var10) [0]) = ((PTR)(1));
     ((var10) [1]) = ((PTR)(7));
     Msg var11 = (Msg)(malloc (8)) ;
     ((var11) [0]) = ((PTR)(1));
     ((var11) [1]) = ((PTR)(var10));
     dlangSignal ((Msg)(chan),(Msg)(var11));
     {Maybe var12 = (Maybe)(malloc (8)) ;
      ((var12) [0]) = ((PTR)(1));
      ((var12) [1]) = ((PTR)(2));
      Msg var13 = (Msg)(malloc (8)) ;
      ((var13) [0]) = ((PTR)(1));
      ((var13) [1]) = ((PTR)(var12));
      dlangSignal ((Msg)(chan),(Msg)(var13));
      {Maybe var14 = (Maybe)(malloc (8)) ;
       ((var14) [0]) = ((PTR)(1));
       ((var14) [1]) = ((PTR)(7));
       Msg var15 = (Msg)(malloc (8)) ;
       ((var15) [0]) = ((PTR)(1));
       ((var15) [1]) = ((PTR)(var14));
       dlangSignal ((Msg)(chan),(Msg)(var15));
       {Maybe var16 = (Maybe)(malloc (8)) ;
        ((var16) [0]) = ((PTR)(1));
        ((var16) [1]) = ((PTR)(3));
        Msg var17 = (Msg)(malloc (8)) ;
        ((var17) [0]) = ((PTR)(1));
```

```
            ((var17) [1]) = ((PTR)(var16));
            dlangSignal ((Msg)(chan),(Msg)(var17));
            {Maybe var18 = (Maybe)(malloc (8)) ;
             ((var18) [0]) = ((PTR)(1));
             ((var18) [1]) = ((PTR)(3));
             Msg var19 = (Msg)(malloc (8)) ;
             ((var19) [0]) = ((PTR)(1));
             ((var19) [1]) = ((PTR)(var18));
             dlangSignal ((Msg)(chan),(Msg)(var19));
             {Msg var20 = (Msg)(malloc (8)) ;
              ((var20) [0]) = ((PTR)(1));
              ((var20) [1]) = ((PTR)(&tag_Nothing));
              dlangSignal ((Msg)(chan),(Msg)(var20));
              return (&tag_Unit);}}}}}}}}}}
List sink (List acc,Msg chan)
  {Maybe v = recvMsg ((Msg)(chan)) ;
   Maybe var0 = v ;
   switch ((int)((var0) [0]))
     {case 0 :
        {return (acc);}
      case 1 :
        {PTR x = (PTR)((var0) [1]) ;
         {List var1 = (List)(malloc (12)) ;
          ((var1) [0]) = ((PTR)(0));
          ((var1) [1]) = ((PTR)(x));
          ((var1) [2]) = ((PTR)(acc));
          return (sink ((List)(var1),(Msg)(chan)));}}}}
Unit initKernelC (int numthreads,void** kstate)
  {KernelState var0 = (KernelState)(malloc (16)) ;
   ((var0) [0]) = ((PTR)(0));
   ((var0) [1]) = ((PTR)(&tag_Nothing));
   ((var0) [2]) = ((PTR)(&tag_Null));
   ((var0) [3]) = ((PTR)(&tag_Null));
   (kstate[(int)(0)]) = (var0);
   return (&tag_Unit);}
Maybe dequeue (int dummy,void** kstate)
  {KernelState ks = kstate[(int)(0)] ;
   KernelState var0 = ks ;
   switch ((int)((var0) [0]))
     {case 0 :
        {Maybe wait = (Maybe)((var0) [1]) ;
         List rtr = (List)((var0) [2]) ;
         List fifo = (List)((var0) [3]) ;
         {List var1 = rtr ;
          switch ((int)((var1) [0]))
            {case 1 :
               {return (&tag_Nothing);}
             case 0 :
               {PTR r = (PTR)((var1) [1]) ;
                List rest = (List)((var1) [2]) ;
                {KernelState var2 = (KernelState)(malloc (16)) ;
                 ((var2) [0]) = ((PTR)(0));
                 ((var2) [1]) = ((PTR)(wait));
                 ((var2) [2]) = ((PTR)(rest));
                 ((var2) [3]) = ((PTR)(fifo));
                 Maybe var3 = (Maybe)(malloc (8)) ;
                 ((var3) [0]) = ((PTR)(1));
```

```
                ((var3) [1]) = ((PTR)(r));
                (kstate[(int)(0)]) = (var2);
                return (var3);}}}}}}}
Unit msgHandler (int tid,int chan,Msg msg,void** kstate)
  {Msg var0 = msg ;
   switch ((int)((var0) [0]))
     {case 1 :
        {Maybe value = (Maybe)((var0) [1]) ;
         return (handleSend ((int)(tid),(Msg)(msg),(void**)(kstate)));}
      case 0 :
        {return (handleRecv ((int)(tid),(Msg)(msg),(void**)(kstate)));}}}
Unit handleSend (int sendTid,Msg msg,void** kstate)
  {{KernelState ks = kstate[(int)(0)] ;
    KernelState var0 = ks ;
    switch ((int)((var0) [0]))
      {case 0 :
         {Maybe wait = (Maybe)((var0) [1]) ;
          List rtr = (List)((var0) [2]) ;
          List fifo = (List)((var0) [3]) ;
          {Maybe var1 = wait ;
           switch ((int)((var1) [0]))
             {case 0 :
                {{Msg var2 = msg ;
                  switch ((int)((var2) [0]))
                    {case 1 :
                       {Maybe val = (Maybe)((var2) [1]) ;
                        {AThread var3 = (AThread)(malloc (12)) ;
                         ((var3) [0]) = ((PTR)(0));
                         ((var3) [1]) = ((PTR)(sendTid));
                         ((var3) [2]) = ((PTR)(&tag_RspAck));
                         List var4 = (List)(malloc (12)) ;
                         ((var4) [0]) = ((PTR)(0));
                         ((var4) [1]) = ((PTR)(var3));
                         ((var4) [2]) = ((PTR)(rtr));
                         KernelState var5 = (KernelState)(malloc (16)) ;
                         ((var5) [0]) = ((PTR)(0));
                         ((var5) [1]) = ((PTR)(&tag_Nothing));
                         ((var5) [2]) = ((PTR)(var4));
                         ((var5) [3]) = ((PTR)(consEOL ((Maybe)(val),
                                                        (List)(fifo))));
                         (kstate[(int)(0)]) = (var5);
                         &tag_Unit;}
                        break;}}}
                 break;}
              case 1 :
                {PTR thd = (PTR)((var1) [1]) ;
                 {AThread var6 = thd ;
                  switch ((int)((var6) [0]))
                    {case 0 :
                       {int recvTid = (int)((var6) [1]) ;
                        Msg x = (Msg)((var6) [2]) ;
                        scheduleSendRecv ((int)(sendTid), (Msg)(msg),
                                 (int)(recvTid), (void**)(kstate));
                        break;}}}
                 break;}}}
          break;}}
    return (&tag_Unit);}}
```

```
Unit handleRecv (int recvTid,Msg msg,void** kstate)
  {{KernelState ks = kstate[(int)(0)] ;
    KernelState var0 = ks ;
    switch ((int)((var0) [0]))
      {case 0 :
         {Maybe wait = (Maybe)((var0) [1]) ;
          List rtr = (List)((var0) [2]) ;
          List fifo = (List)((var0) [3]) ;
          {Maybe var1 = wait ;
           switch ((int)((var1) [0]))
             {case 0 :
                {{List var2 = fifo ;
                  switch ((int)((var2) [0]))
                    {case 1 :
                       {{AThread var3 = (AThread)(malloc (12)) ;
                         ((var3) [0]) = ((PTR)(0));
                         ((var3) [1]) = ((PTR)(recvTid));
                         ((var3) [2]) = ((PTR)(msg));
                         Maybe var4 = (Maybe)(malloc (8)) ;
                         ((var4) [0]) = ((PTR)(1));
                         ((var4) [1]) = ((PTR)(var3));
                         KernelState var5 = (KernelState)(malloc (16)) ;
                         ((var5) [0]) = ((PTR)(0));
                         ((var5) [1]) = ((PTR)(var4));
                         ((var5) [2]) = ((PTR)(rtr));
                         ((var5) [3]) = ((PTR)(fifo));
                         (kstate[(int)(0)]) = (var5);
                         &tag_Unit;}
                        break;}
                     case 0 :
                       {PTR hd = (PTR)((var2) [1]) ;
                        List tl = (List)((var2) [2]) ;
                        {Msg var6 = (Msg)(malloc (8)) ;
                         ((var6) [0]) = ((PTR)(3));
                         ((var6) [1]) = ((PTR)(hd));
                         AThread var7 = (AThread)(malloc (12)) ;
                         ((var7) [0]) = ((PTR)(0));
                         ((var7) [1]) = ((PTR)(recvTid));
                         ((var7) [2]) = ((PTR)(var6));
                         List var8 = (List)(malloc (12)) ;
                         ((var8) [0]) = ((PTR)(0));
                         ((var8) [1]) = ((PTR)(var7));
                         ((var8) [2]) = ((PTR)(rtr));
                         KernelState var9 = (KernelState)(malloc (16)) ;
                         ((var9) [0]) = ((PTR)(0));
                         ((var9) [1]) = ((PTR)(&tag_Nothing));
                         ((var9) [2]) = ((PTR)(var8));
                         ((var9) [3]) = ((PTR)(tl));
                         (kstate[(int)(0)]) = (var9);
                         &tag_Unit;}
                        break;}}}
                 break;}
               case 1 :
                 {PTR thd = (PTR)((var1) [1]) ;
                  {AThread var10 = thd ;
                   switch ((int)((var10) [0]))
                     {case 0 :
```

```
                       {int sendTid = (int)((var10) [1]) ;
                        Msg sendMsg = (Msg)((var10) [2]) ;
                        scheduleSendRecv
((int)(sendTid),(Msg)(sendMsg),(int)(recvTid),(void**)(kstate));
                        break;}}}
                 break;}}}
          break;}}
    return (&tag_Unit);}}
Unit scheduleSendRecv (int sendTid,Msg sendMsg,int recvTid,void** kstate)
  {Msg var0 = sendMsg ;
   switch ((int)((var0) [0]))
     {case 1 :
        {Maybe value = (Maybe)((var0) [1]) ;
         {AThread var1 = (AThread)(malloc (12)) ;
          ((var1) [0]) = ((PTR)(0));
          ((var1) [1]) = ((PTR)(sendTid));
          ((var1) [2]) = ((PTR)(&tag_RspAck));
          Msg var2 = (Msg)(malloc (8)) ;
          ((var2) [0]) = ((PTR)(3));
          ((var2) [1]) = ((PTR)(value));
          AThread var3 = (AThread)(malloc (12)) ;
          ((var3) [0]) = ((PTR)(0));
          ((var3) [1]) = ((PTR)(recvTid));
          ((var3) [2]) = ((PTR)(var2));
          List var4 = (List)(malloc (12)) ;
          ((var4) [0]) = ((PTR)(0));
          ((var4) [1]) = ((PTR)(var3));
          ((var4) [2]) = ((PTR)(&tag_Null));
          List var5 = (List)(malloc (12)) ;
          ((var5) [0]) = ((PTR)(0));
          ((var5) [1]) = ((PTR)(var1));
          ((var5) [2]) = ((PTR)(var4));
          KernelState var6 = (KernelState)(malloc (16)) ;
          ((var6) [0]) = ((PTR)(0));
          ((var6) [1]) = ((PTR)(&tag_Nothing));
          ((var6) [2]) = ((PTR)(var5));
          ((var6) [3]) = ((PTR)(&tag_Null));
          (kstate[(int)(0)]) = (var6);
          return (&tag_Unit);}}}}
List consEOL (PTR value,List lst)
  {List var0 = lst ;
   switch ((int)((var0) [0]))
     {case 1 :
        {{List var1 = (List)(malloc (12)) ;
          ((var1) [0]) = ((PTR)(0));
          ((var1) [1]) = ((PTR)(value));
          ((var1) [2]) = ((PTR)(&tag_Null));
          return (var1);}}
      case 0 :
        {PTR hd = (PTR)((var0) [1]) ;
         List tl = (List)((var0) [2]) ;
         {List var2 = (List)(malloc (12)) ;
          ((var2) [0]) = ((PTR)(0));
          ((var2) [1]) = ((PTR)(hd));
          ((var2) [2]) = ((PTR)(consEOL ((PTR)(value),(List)(tl))));
          return (var2);}}}}
void*  thread_1 (void *args)
```

```
  {Msg src2encoder = ((void**)args)[0] ;
   {return (src ((Msg)(src2encoder)));}}
void*  thread_2 (void *args)
  {Msg decoder2sink = ((void**)args)[0] ;
   {return (sink ((List)(&tag_Null),(Msg)(decoder2sink)));}}
void*  thread_3 (void *args)
  {Msg src2encoder = ((void**)args)[0] ;
   Msg encoder2decoder = ((void**)args)[1] ;
   {return (encode ((HuffmanTree)(tree1
()),(Msg)(src2encoder),(Msg)(encoder2decoder)));}}
void*  thread_4 (void *args)
  {Msg encoder2decoder = ((void**)args)[0] ;
   Msg decoder2sink = ((void**)args)[1] ;
   {return (decode ((HuffmanTree)(tree1 ()),(HuffmanTree)(tree1
()),(Msg)(encoder2decoder),(Msg)(decoder2sink)));}}
int main ()
  {Msg src2encoder = createService (2,&msgHandler,&dequeue,&initKernelC) ;
   Msg decoder2sink = createService (2,&msgHandler,&dequeue,&initKernelC)
;
   Msg encoder2decoder = createService
(2,&msgHandler,&dequeue,&initKernelC) ;
   pthread_t _pt_thread_1 ;
   pthread_t _pt_thread_2 ;
   pthread_t _pt_thread_3 ;
   pthread_t _pt_thread_4 ;
   dlang_thread_create (&_pt_thread_1,thread_1,1,src2encoder);
   dlang_thread_create (&_pt_thread_2,thread_2,1,decoder2sink);
   dlang_thread_create
(&_pt_thread_3,thread_3,2,src2encoder,encoder2decoder);
   dlang_thread_create
(&_pt_thread_4,thread_4,2,encoder2decoder,decoder2sink);
   pthread_join (_pt_thread_1,NULL);
   pthread_join (_pt_thread_2,NULL);
   pthread_join (_pt_thread_3,NULL);
   pthread_join (_pt_thread_4,NULL);
   return (0);}
```

Appendix C – Rosetta Specifications

This contains additional Rosetta specifications developed during the feasibility study.

C.1 Advanced Encryption Standard (AES)

This specification represents an early attempt to capture an important radio function, encryption. As a preliminary specification, it explores a number of approaches to defining the AES function. Further refinement is in order.

```
//----
// AES Encryption Specification
//
// Author: Perry Alexander
// Date: Wed Sep 12 14:20:30 CDT 2007
// Revision:
//  Thu Jul 24 12:05:24 CDT 2008 - wpa -- added initialization to the
//  AES facet.
//  Tue Sep 25 15:00:13 CDT 2007 - wpa - first reasonable release
//
// Todo:
// - Must define the sbox constant value
// - Must define the rcon constant value
// - Define rcon iteration in expand key
// - Redefine subBytes with subBytesRow
//
// Basic AES algorithm
//
// AES(state,cipherkey)
//   KeyExpansion(cipherkey,expandedkey)
//   addroundkey(state, expandedkey)
//   for (i=1; i<Nr; i++){
//     round(state,expandedkey + Nb*i)}
//   finalround(state,expandkey + Nb * Nr)
//
//----

package aes()::state_based is
    export AES;
    ////--- Basic Types and Constants

    // Define a type for byte
    byte :: type is word(8);

    // Define a types for a 4 byte row and a 4x4 byte array
    Rowtype :: type is sel(r::sequence(byte) | #r = 4);
    blockType :: type is sel(b::sequence(rowType) | #b = 4);

    // Byte rotate a word
    rotate(a::rowType)::rowType is rotl(a);

    // Grab a column as a sequence
    column(x::{0,1,2,3}; a::blockType)::rowType is
    [ a(0)(x),a(1)(x),a(2)(x),a(3)(x) ];

    // Declare the sbox constant array
    sboxType :: type is sel(x::sequence(byte) | #x=16);
    sbox :: sboxType is constant;

    // Declare the rcon constant array
    rconType :: type is sel(x::sequence(byte) | #x=256);
    rcon :: rconType is constant;

    ////---- Basic Encryption Utility Functions

    // Define addRoundKey as zipped xor
    addRoundRow(a,k::rowType)::rowType is zip(xor,a,k);
```

```
    addRoundKey(a,k::blockType)::blockType is zip(addRoundRow,a,k);

    // Define shiftRow using explicit sequence construction
    shiftRow(a::blockType)::blockType is
    [
        a(0),
        rotl(a(1)),
        rotl(rotl(a(2))),
        rotl(rotl(rotl(a(3))))
    ];

    // Define subBytes using properties.
    // Each byte in the result is defined as the value in sbox indexed by
    // the value in a.
    subBytes(a::blockType)::blockType
    where forall(i::{0,1,2,3} |
                   forall(j::{0,1,2,3} | subBytes(a)(i)(j) =
                                          sbox(bv2nat(a(i)(j)))));

    // Define utility function subBytesRow using explicit
    //   sequence creation
    subBytesRow(a::rowType)::rowType is
      [sbox(bv2nat(a(0))),
       sbox(bv2nat(a(1))),
       sbox(bv2nat(a(2))),
       sbox(bv2nat(a(3)))];

    // Multiply a row from a by a column from b
    mkElem(y::{0,1,2,3};a,b::blockType)::byte is
rowMult(a(x),column(b,y));

    // Define mixColumn as matrix multiplication in the classical
    //   style.  mkElem
    // calcuates an element of the product.  Uses explicit matrix creation
    mixColumn(a::blockType)::blockType is
    let b::blockType be [[x"02",x"03",x"01",x"01"],
                         [x"01",x"02",x"03",x"01"],
                         [x"01",x"01",x"02",x"03"],
                         [x"03",x"01",x"01",x"02"]] in

[[mkElem(0,0,a,b),mkElem(0,1,a,b),mkElem(0,2,a,b),mkElem(0,3,a,b)],

[mkElem(1,0,a,b),mkElem(1,1,a,b),mkElem(1,2,a,b),mkElem(1,3,a,b)],

[mkElem(2,0,a,b),mkElem(2,1,a,b),mkElem(2,2,a,b),mkElem(2,3,a,b)],

[mkElem(3,0,a,b),mkElem(3,1,a,b),mkElem(3,2,a,b),mkElem(3,3,a,b)]]
    end let;

    // Define multiplication of two rows to get an element.  Use xor
    // as addition defined by the AES specifiation
    rowMult(a,b :: rowType)::byte is
       rmult(a(0),b(0))
       xor rmult(a(1),b(1))
       xor rmult(a(2),b(2))
       xor rmult(a(3),b(3));
```

```
// Multiply by 1 or zero.  Basically a word and.
bitMult(y::byte,x::bit)::byte is
   map(<*(b::bit)::bit is b and x*>,y);

// Generate a single minterm in the multiply
multTerm(x,b,y)::word(16) is
   lshl(x) xor bitMult(b,y);

// Multiply two bytes
byteMult(x,y::byte)::word(16) is
   reduce(multTerm(y),x"0000",x);

// Define Rijndael's multiplication for bytes using byte multiply
// Could also be a lookup table.  This function should be validated
// with the AES specification if this actually gets used
rmult(a,b :: byte)::byte is
  let x::word(16) be byteMult(a,b) in
x sub [15,..8] xor x sub [7,..0];

// Define Key Schedule function using replace and rotate.
//   subBytes row using same replacement array as subBytes.
// There are two definitions
//   for this in the documentation.  This is the second called
//   core
core(a::rowType; i::natural)::rowType is
let s::rowType be subBytesRow(rotate(a)) in
    replace(s,0,s(0) xor rcon(i))
end let;

//// Key Expansion

// Key sizes are in bytes and must be 16,24,34 - fix this type earlier
// Compare key size with expanded key size to control recursion
// For 256 bit keys, there is an extra sbox lookup that
//   is not included in this specification
// I am quite skeptical that this function is completely correct
// Must define rconIteration which is an index into the rcon table
expandKey(key::sequence(byte);
          keySize,xKeySize::natural)::sequence(byte) is
if keySize=xKeySize
then key
else
    let t::sequence(byte) be
      core(key sub [#key-4,#key-3,#key-2,#key-1],rconIteration) in
        expandKey(key &
                  zip(xor,(key sub [#key-4,#key-3,#key-2,#key-1]),t),
                  keySize + 4,
                  xKeySize)
    end let;
end if;

//// Utility functions for the encryption facet

// Call the utility functions in sequence as an AES round
aesRound(state::blockType; roundKey::blockType)::blockType is
addRoundKey(mixColumns(shiftRows(subBtyes(state))),roundKey);
```

```
 // Create a round key from a sequence of bytes
 createRoundKey(xk::sequence(byte))::blockType is
 [ [ xk(0),xk(4),xk(8),xk(12) ]
     [ xk(1),xk(5),xk(9),xk(13) ]
     [ xk(2),xk(6),xk(10),xk(14) ]
     [ xk(3),xk(7),xk(11),xk(15) ] ];

 ////
 //// AES Encryption Facet
 // This is a state-based facet whose state is an encryption block.
 // round counts the number of rounds for a block.  Round 0 loads a
 // block from input and subsequent rounds apply the
 // encryption algorithm.
 // Round nbrRounds - 1 is the last round for a block and subsequently
 // outputs the encryption state.
 //
 // Assumptions:
 // - There is no input ready signal. The device pulls input when it is
 //   ready for input.
 // - If the key length is not 16, 24, or 32 bytes,
 //     behavior is not defined.
 // - expandedKey is constant for each instance of the facet.  Move the
 //   initialization to a term to vary the key
 // - What drives state change is not defined.
 //     Refine to a different domain to specify.
 // - Make sure the interface and body syntax matches
 //     current specifications
 facet interface AES(din::input sequence(byte);
                     dout::output sequence(byte);
                     key::input sequence(byte);
                     reset::input boolean;
                     done::output boolean;
                     size,xsize::design natural) ::
                      state_based(blockType)
 end facet interface AES;

 facet body AES is
     // Expand the key
     expandedKey :: sequence(byte) is expandKey(key,size,xsize);
     // Calculate counter from key length
     nbrRounds :: natural is if #key = 16 then 10
                             elseif #key = 24 then 12
                             elseif #key = 32 then 14
                             end if;
     // encryption round counter
     round :: natural;
// input buffer
buff :: sequence(byte);
 begin
     // Update the round counter after each step.
     //   If reset is true, then set the round to 0.
     nextRound: round' = if reset then 0
                 elseif round<nbrRounds then round+1
                      else 0
                      end if;
     // Create the round key for each step
     nextKey: roundKey' = createRoundKey(expandedKey sub
```

```
                                          [round,..round+15]);
        // Latch the inputs to make sure they do not change
        //   during the rounds
        latch: buff' = if round'=0 then din else buff end if;
        // Either initialize the state, or move to the next state
        //   based on the counter.
        nextState: s' = if round = 0
                        then aesRound([din sub [0,..3],
                                       din sub [4,..7],
                                       din sub [8,..11],
                                       din sub [12,..15]],roundKey)
                        else aesRound(s,roundKey)
                        end if;
        // Output the block after the last round.
        //   Hold output constant until last round is processed.
        nextOut: dout' = if round = nbrRounds - 1
                         then s(0) & s(1) & s(2) & s(3)
                         else dout
                         end if;
        // Output a done signal when the encryptor is done with one block
   doneOut: done' = if round = nbrRounds -1;
    end facet body AES;

end package aes;
```

C.2 Standard Modulation Specifications

Like the AES specification, this specification of modulation functions attempts to capture an initial definition of the variety of modulation functions. Further refinement is in order.

```
//----
// Modulator Package
//
// Author: Perry Alexander
// Date: Mon Dec 17 18:17:11 CST 2007
// Revision:
//   Thu Jun 19 23:42:00 CST 2008 -- Separated the function
//     of the modulator from
//   the underlying state sequencing mechanism.
//   Wed Jun 18 12:47:00 CST 2008 -- Added a modulator
//     domain that partially instantiates the async domain.
//   Mon Dec 17 22:39:54 CST 2007 -- Added binary encodings
//     to accompany quad encodings
//   Tue Dec 18 17:09:30 CST 2007 -- Added new keying functions
//     and removed general quad and binary structures.
//     Worked through some other comm issues
//   Thu Dec 20 14:33:09 CST 2007 -- Added new keying functions
//     for 16-value qam, qfm, and qpsk modulation.
// Todo:
//   - The instantiation technique used for defining the modulator domain
//     needs to be checked.
//   - New keying functions need to be sanity checked
//   - Further parameterize keying and modulation functions to allow for
//     arbitrary M (where M is a power of 2).
//   - MQAM still needs to be defined unless MQAM is what I'm calling QAM
//     below.
//----

package modulator()::static is

    // Constant values

    twoPi :: real is 2*pi;
    sqrt2 :: real is abs(sqrt(2));

    // Basic parameterized modulation function.  am, fm, and ps are the
    // amplitude multiplier, frequency shift and phase shift respectively.
    // Set ip to false to include a 90 degree phase shift.
    // f is the carrier frequency and t is time.

    modulate(ip::boolean; am,fm,ps::real; f,t::real)::real is
      if ip then am*sin(twoPi*(f+fm)*t+ps)
      else am*cos(twoPi*(f+fm)*t+ps)
      end if;

    // Basic modulation functions for AM, FM and PSK.  k is the keying
    // function that translates the input symbol to a
    // real value for modulation.
    // s is the input symbol to be modulated.  Thus, k(s) gives the value
    // to be modulated.
    // ip, f, and t play the same role as in the modulate
    // function - 90 phase shift, carrier frequency, and time.
    // The universally quantified type, T, is the input type that
    //   will not be known until the function is instantiated.

    amMod[T::type](k::<*(x::T)::real*>; ip::boolean;
          f::real; t::real; s::T)::real is
```

```
        modulate(ip,k(s),0.0,0.0,f,t);

fmMod[T::type](k::<*(x::T)::real*>; ip::boolean;
      f::real; t::real; s::T)::real is
        modulate(ip,1.0,k(s),0.0,f,t);

pskMod[T::type](k::<*(x::T)::real*>; ip::boolean;
      f::real; t::real; s::T)::real is
        modulate(ip,1.0,1.0,k(s),f,t);

// Example key functions for quadrature style modulation.  kam, kfm
// kpsk are for qam, qfm, and qpsk modulation respectively.

kam(b::bit)::real is b;
kfm(b::bit)::real is if %b then 5e3 else 0 end if;
kpsk(b::bit)::real is if %b then pi else -pi end if;

// Modulator functions for specific quatrature modulation schemes
// defined using basic modulation functions

qamMod(f,t::real;s::word(2))::real is
  amMod(kam,true,f,t,s(0))+amMod(kam,false,f,t,s(1));

qfmMod(f,t::real;s::word(2))::real is
  fmMod(kfm,true,f,t,s(0))+fmMod(kfm,false,f,t,s(1));

qpskMod(f,t::real;s::word(2))::real is
  pskMod(kpsk,true,f,t,s(0))+pskMod(kpsk,false,f,t,s(1));

// Modulation facets defining discrete time modulation
//   components.  Note that outputs are real and inputs are
//   two-bit sequences.  All modulators are currently defined
//   in discrete time, but could easily be defined in
//   continuous time. The reference to time in the modulation
//   functions makes simple state-based specifications difficult
//   to write without specifying a type for state.

// A modulator component is an state_based component with
//   an additional design parameter to set the carrier frequency.
//   Thus, the modulator function domain is simply the
//   discrete_time domain with an input and output, plus a
//   design parameter for specifying carrier frequency.
//   We choose discrete_time over state_based because a time
//   value is required for the modulation function.
domain modulator_function
    [D,R::type]
    (i::input D; o::output R; f::design real)::discrete_time is
begin
end domain modulator_function;

// Define functional specifications for each modulator type
//   without regard to the underlying control model.
//   Note that t is defined by the discrete_time domain.
facet qam_mod_function::modulator_function is
begin
  o' = en*qamMod(f,t,i);
end facet qam_mod_function;
```

```
facet qfm_mod_function::twoPhase is
begin
    o' = en*qfmMod(f,t,i);
end facet qfm_mod_function;

facet qpsk_mod_function::twoPhase is
begin
    o' = en*qpskMod(f,t,i);
end facet qpsk_mod_function;

// To define a modulator using a specific underlying communication
// architecture, form the product of the modulator function with a
// communication strategy.  For example, the async package defines
// a strategy called asyncMinus that only defines state sequencing

// async_comm adds nothing to the asyncMinus definition.  It can, but
// for this initial example, there is no need to do so.
facet modulator_control::asyncMinus is
begin
end facet modulator_control;

// Now define some facet compositions that make modulators
//   asynchronous. The asynchronous modulator components are
//   the conjunction of the modulator function and the underlying
//   control architecture.
qam_mod_async
     [D,R::type]
     (i::input D;
      o::output R;
      go::input bit;
      ready::input bit)::discrete_time is
  qam_mod_function[D,R::type](i::input D; o::output R; f::real) *
  modulator_control(go::input bit; ready::input bit);

qfm_mod_async
     [D,R::type]
     (i::input D;
      o::output R;
      go::input bit;
      ready::input bit)::discrete_time is
  qfm_mod_function[D,R::type](i::input D; o::output R; f::real) *
  modulator_control(go::input bit; ready::input bit);

qpsk_mod_async
     [D,R::type]
     (i::input D;
      o::output R;
      go::input bit;
      ready::input bit)::discrete_time is
  qpsk_mod_function[D,R::type](i::input D; o::output R; f::real) *
  modulator_control(go::input bit; ready::input bit);

// An alternate aproach would combine the modulator
//   domain with the asynchronous domain at the domain level.

// BAM, BFM, and BPSK can be defined similarly using the same keying
```

```
// functions and the same modulation functions.


bamMod(f,t::real; s::bit)::real is amMod(kam,true,f,t,s);

bfmMod(f,t::real; s::bit)::real is fmMod(kfm,true,f,t,s);

bpskMod(f,t::real; s::bit)::real is pskMod(kpsk,true,f,t,s);

// Modulation facets for binary techniques follow similarly

facet bam_mod_function::modulator_function is
begin
    o' = en*bamMod(f,t,i);
end facet bam_mod_function;

facet bfm_mod_function::modulator_function is
begin
    o' = en*bfmMod(f,t,i);
end facet bfm_mod_function;

facet bpsk_mod_function::modulator_function is
begin
    o' = en*bpskMod(f,t,i);
end facet bpsk_mod_function;

bam_mod_async
     [D,R::type]
     (i::input D;
      o::output R;
      go::input bit;
      ready::input bit)::discrete_time is
  bam_mod_function[D,R::type](i::input D; o::output R; f::real) *
  modulator_control(go::input bit; ready::input bit);

bfm_mod_async
  [D,R::type]
     (i::input D;
      o::output R;
      go::input bit;
      ready::input bit)::discrete_time is
  bfm_mod_function[D,R::type](i::input D; o::output R; f::real) *
  modulator_control(go::input bit; ready::input bit);

bpsk_mod_async
     [D,R::type]
     (i::input D;
      o::output R;
      go::input bit;
      ready::input bit)::discrete_time is
  bpsk_mod_function[D,R::type](i::input D; o::output R; f::real) *
  modulator_control(go::input bit; ready::input bit);

// Allowing more bits in the input symbol requires different keying
// functions.

angle(sc::natural)::real is twoPi/sc;
```

```
    // Define some constant key functions for the various modulation
    // schemes.  Note that these have not been checked!!

    kamX(i::word(4))::real is sin(bv2nat(i)*angle(16));
    kamY(i::word(4))::real is cos(bv2nat(i)*angle(16));
    kfmX(i::word(4))::real is sin(bv2nat(i)*angle(16))*5e3;
    kfmY(i::word(4))::real is cas(bv2nat(i)*angle(16))*5e3;
    kpskX(i::word(4))::real is sin(bv2nat(i)*angle(16))*pi;
    kpskY(i::word(4))::real is sin(bv2nat(i)*angle(16))*pi;

    // Modulator functions for specific quatrature modulation schemes
    // defined using basic modulation functions

    amMod16(f,t::real;s::word(2))::real is
      amMod(kamX,true,f,t,s sub [0,1])+amMod(kamY,false,f,t,s sub [2,3]);

    fmMod16(f,t::real;s::word(2))::real is
      fmMod(kfmX,true,f,t,s sub [0,1])+fmMod(kfmY,false,f,t,s sub [2,3]);

    pskMod16(f,t::real;s::word(2))::real is
      pskMod(kpskX,true,f,t,s sub [0,1])+pskMod(kpskY,false,f,t,s sub
[2,3]);

    // These can be further parameterized to perform m-ary modulation
    //   as longas the constellations fit the schemes defining by keying
functions.  It
    // is quite simple to define additional keying functions for different
    // constellation shapes.

end package modulator;
```

List of Acronyms, Abbreviations, and Symbols

Acronym	Description
AES	Advanced Encryption Standard
AM	Amplitude Modulation
ASIC	Application Specific Integrated Circuit
ASK	Amplitude Shift Modulation
AST	Abstract Syntax Tree
BPSK	Binary Phase Shift Modulation
CORBA	Common Object Request Broker Architecture
CW	Continuous Wave Modulation
DSP	Digital Signal Processor
FM	Frequency Modulation
FPGA	Field Programmable Gate Array
GPP	General Purpose Processor
JTRS	Joint Tactical Radio Systems
PSK	Phase Shift Modulation
QAM	Quadrature Amplitude Modulation
QPSK	Quadrature Phase Shift Modulation
SCA	Software Communications Architecture\
SDR	Software Defined Radio
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuits